POLITÉCNICO DE LEIRIA
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Lean-Inspired QA and CI/CD Re-Engineering Towards AI-Ready Infrastructures

A Case Study in Workflow, Test Architecture, and Pipeline Restructuring at a SaaS Company

José Pedro Nolasco Henriques

School of Management and Technology

Department of Computer Engineering

Master in Computer Engineering - Mobile Computing

Leiria, May 2026

This page intentionally left blank.

POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Lean-Inspired QA and CI/CD Re-Engineering Towards AI-Ready Infrastructures

A Case Study in Workflow, Test Architecture, and Pipeline Restructuring at a SaaS Company

José Pedro Nolasco Henriques

Student No. 2242699

Supervisor: Ricardo Jorge Pereira Gomes
Full Professor, Polytechnic of Leiria

School of Management and Technology

Department of Computer Engineering

Master in Computer Engineering - Mobile Computing

Internship

Leiria, May 2026

This page intentionally left blank.

Lean-Inspired QA and CI/CD Re-Engineering Towards AI-Ready Infrastructures

Preparation of this work was facilitated by the use of the *IPLeiria-Thesis* template.

This page intentionally left blank.

Acknowledgements

First of all, I want to thank Professor Ricardo Dias, for the outstanding guidance, availability and persistent support that went beyond helping write this document. I am deeply grateful for what I was able to learn from him and with him, inside and outside the engineering context. His knowledge always seemed limitless and it was a uniquely great experience to be able to have his company for so long, one that I'm sure I'll remember deeply. Secondly, I want to thank Glartek for the possibility of Internship that led to this document being built and the conclusion of my studies, with a special acknowledgment to Cláudio and Pedro, my two seniors from whom I was able to learn a lot and who allowed me to participate actively within the company environment, being able to learn and grow so much. To them, I am also extremely grateful, being sure that they made a great difference in the quality of the moments and lessons learned there. I'd like to thank my close family, my dear parents and sisters for always supporting me through my studies and the incentive to pursue this master's degree, always lending a shoulder when needed, even in the more hopeless moments. My closest friends, whom I shared so many moments and lived so many things, with always something new to learn, or simply a place to free the mind from the noise of life. Lastly and perhaps most importantly, I would like to thank my girlfriend, Magdalena. She who always gave her wholehearted best and lived every moment of this journey with me, celebrating the best moments and supporting me on the worst. To her, who moved to Portugal to be with me as I completed my studies, even though it meant moving away from her family and friends, I do not have words that would be enough to describe my gratitude or could ever match the size of her gesture. I thank her from the bottom of my heart for always staying by my side, being my pride and my biggest source of awe.

This page intentionally left blank.

Resumo

Este relatório descreve um estágio de Engenharia de Software na área de Garantia de Qualidade na Glartek, uma startup portuguesa de SaaS posicionada como solução nativa de inteligência artificial para a gestão digital de operações em clientes industriais. Na sequência de um período de crescimento acelerado, os processos de QA e de CI/CD da empresa acumularam dívida técnica significativa, traduzida em ciclos de feedback longos, instabilidade na bateria de testes, uso ineficiente de recursos e fronteiras de responsabilidade pouco claras.

O esforço de reengenharia foi orientado pelos princípios do Lean Software Development e organizado em quatro áreas: redesenho do fluxo de trabalho, arquitetura de software e hardware, reestruturação de dados e testes, e suporte a migrações de ferramentas. As intervenções abrangeram a clarificação da atribuição de tarefas, a otimização dos runners, a reestruturação da bateria com uma abordagem baseada em seed e uma pirâmide de testes mais equilibrada, a migração de Cypress para Playwright, e o desenvolvimento de um script de automação do ambiente local e de uma extensão de browser para reporte de feedback.

As intervenções produziram melhorias mensuráveis: a reestruturação da área Actions reduziu o tempo de execução em 43%; a migração para Playwright acrescentou uma redução adicional de 26%, totalizando 58% face ao baseline. A instabilidade global desceu de 520 instâncias de falha por 100 execuções para 203, com a proporção de execuções sem falhas a subir de 16% para 42%. A infraestrutura resultante está posicionada para suportar fluxos de trabalho de testes assistidos por inteligência artificial, alinhados com a direção estratégica da Glartek enquanto plataforma nativa de IA.

Palavras-Chave: QA, CI/CD, Lean Software Development, Inteligência Artificial, Reengenharia de Processos, Automação de Testes

This page intentionally left blank.

Abstract

This report describes a Software Engineering internship at Glartek, a Portuguese SaaS startup positioned as an AI-native digital operations management platform for industrial clients. Following rapid growth, the company's QA and CI/CD processes had accumulated significant technical debt, manifesting as long feedback cycles, test suite instability, inefficient resource usage, and unclear responsibility boundaries between development and QA teams.

Guided by Lean Software Development principles, the re-engineering effort covered four areas: workflow redesign, software and hardware architecture, data and test restructuring, and tooling and migration support. Task ownership was clarified through new labelling conventions; runner usage was optimised; the test suite was restructured with a seed-driven approach and a more balanced testing pyramid; and the end-to-end suite was migrated from Cypress to Playwright alongside a local environment automation script and a feedback recording browser extension.

The interventions produced measurable results: the restructured Actions area achieved a 43% reduction in execution time, with the Playwright migration adding a further 26%, for a cumulative 58% reduction. Battery-wide instability fell from 520 failure instances per 100 runs to 203, with failure-free runs rising from 16% to 42%. The resulting infrastructure is positioned to support AI-assisted testing workflows aligned with Glartek's direction as an AI-native platform.

Keywords: QA, CI/CD, Lean Software Development, Artificial Intelligence, Process Re-Engineering, Test Automation

This page intentionally left blank.

AI Acknowledgement

Example of Use

I acknowledge the use of Cursor (https://cursor.com) and Claude (https://claude.ai) to refine the academic tone and improve the linguistic accuracy of this work, including aspects of grammar, punctuation, and vocabulary.

This page intentionally left blank.

Contents

This page intentionally left blank.

List of Figures

This page intentionally left blank.

List of Tables

This page intentionally left blank.

Glossary

Backlog A prioritised list of pending work items, features, or defects awaiting development or testing. Teams regularly review and reorder the backlog to ensure the most important items are addressed first. (*p. xvi*)

Blue/Green Deployment A release strategy that maintains two identical production environments — one active (blue) and one idle (green). New versions are deployed to the idle environment and traffic is switched over once validated, enabling zero-downtime releases and instant rollback. (*p. xvi*)

Canary Release A deployment technique in which a new version of an application is gradually rolled out to a small subset of users before a full release. This limits the impact of potential defects while still allowing real-world validation. (*p. xvi*)

DevSecOps An extension of DevOps that integrates security practices directly into the development and delivery pipeline. Rather than treating security as a final gate, DevSecOps embeds security checks, testing, and validation throughout the entire software lifecycle. (*p. xvi*)

Feature Flag A software mechanism that enables or disables a feature at runtime without deploying new code. Feature flags allow teams to release code in a disabled state, selectively activate it for specific users or environments, and roll it back instantly if issues arise. (*p. xvi*)

Flakiness The property of a test or pipeline job that produces inconsistent results — passing on some executions and failing on others — without any change to the underlying code. Flaky tests erode trust in the automated suite and introduce unnecessary overhead through false failures and repeated reruns. (*p. xvi*)

Headless Browser A web browser that operates without a visible graphical interface. Headless execution is standard in CI/CD environments because it allows automated browser tests to run on servers where no display is available, while still exercising the full browser rendering and JavaScript engine. (*p. xvi*)

Regression A defect introduced into functionality that previously worked correctly, typically as a side effect of a new change elsewhere in the codebase. Regression testing aims to detect these unintended breakages before a release reaches production. (*p. xvi*)

Runner An agent — either a physical machine, virtual machine, or container — that picks up and executes jobs defined in a CI/CD pipeline. In GitLab CI, runners are registered to a project and process jobs in the order and conditions specified by the pipeline configuration. (*p. xvi*)

Seed Data A predefined, controlled dataset loaded into a test environment before test execution begins. Using seed data instead of creating entities through the application interface during each test run reduces setup time, lowers resource consumption, and produces more deterministic and reproducible test conditions. (*p. xvi*)

Shift-Left A quality and security philosophy that moves testing and validation activities to earlier stages of the development lifecycle. By catching defects closer to their source — during development rather than after integration — shift-left practices reduce the cost and effort of corrections and shorten feedback cycles. (*p. xvi*)

Technical Debt The accumulated cost of shortcuts, suboptimal design decisions, and deferred improvements made during software development. Like financial debt, technical debt accrues interest: the longer it goes unaddressed, the harder and more expensive it becomes to work around or resolve. (*p. xvi*)

This page intentionally left blank.

Acronyms

AI Artificial Intelligence. (*p. xvi*)
API Application Programming Interface. (*p. xvi*)
AR Augmented Reality. (*p. xvi*)

CD Continuous Delivery. (*p. xvi*)
CI Continuous Integration. (*p. xvi*)

E2E End-to-End. (*p. xvi*)

IaC Infrastructure as Code. (*p. xvi*)
IoT Internet of Things. (*p. xvi*)

MR Merge Request. (*p. xvi*)

QA Quality Assurance. (*p. xvi*)

R&D Research and Development. (*p. xvi*)
RTS Regression Test Selection. (*p. xvi*)

TBTS Transition-Based Test Selection. (*p. xvi*)
TCP Test Case Prioritisation. (*p. xvi*)

UI User Interface. (*p. xvi*)

This page intentionally left blank.

1

Introduction

Every story has a beginning, and this one starts at Glartek, a startup based in Parceiros, Leiria. Established by Luís Murcho in 2017, Glartek was founded with the purpose of revolutionizing how frontline workers connect with operations, creating strong ties between on-site and back-office teams. The concept was widely received by the public, earning the company numerous awards, including "Best Portuguese Startup" at Web Summit 2017 and the EDP Innovation Award.

Initially, the company focused on providing guidance for industrial processes using Augmented Reality (AR) and the Internet of Things (IoT) to improve operational security, performance, and quality. This concept matured into a mobile-first, no-code Environment, Health, Safety, and Quality (EHSQ) Connected Worker platform and a back-office system accessible via mobile and web applications that digitizes operational processes into structured, executable workflows. Here, every task is integrated with step-by-step instructions and contextual details, creating guided procedures that ensure proper task execution and quality assurance. The platform also consolidates information collected during these processes — data that was previously scattered across various employees, contexts, and archaic paper formats — into a single digital ecosystem, driving safety, performance, and quality across every process.

For clients ranging from energy companies like Finerge and EDP, to industrial manufacturers such as BOSCH and SIEMENS, and beverage producers like Sumol+Compal, Glartek heavily reduces paperwork and decreases incidents while delivering faster onboarding for new employees.

Recently, the company shifted its strategy towards comprehensively supporting the worker by facilitating the documentation and management of all process-related tasks and events. Straying slightly from the initial idea of a pure "in-task guiding role," Glartek has increased its leverage in process security by developing features such as event occurrence registries and work permits. The company is now positioning itself

as an AI-native EHSQ solution, integrating modern artificial intelligence technologies into the platform to further enhance user experience, automate compliance workflows, and strengthen service quality.

This brings us to the present: Glartek is experiencing fast and successful growth, accompanied by a surge in clients and an exponential increase in employees to keep up with the demand for new features and platform maintenance. However, this growth has presented the challenge of scaling infrastructure and processes to meet demand while maintaining the quality of service.

This report presents the results of work as a Software Engineer Intern in Quality Assurance at Glartek, responsible for assuring the quality of developed features and improving the efficiency, security, and quality of existing processes.

1.1 Motivation

As detailed in the previous section, Glartek has recently experienced a period of exponential growth, characterized by an expanding client base and a rapidly increasing team size. Previously, with a smaller team to keep pace with market demands, the development philosophy naturally prioritized speed of delivery and feature implementation.

While this "delivery-first" approach was crucial for the company's initial success and market capture, it inadvertently led to the accumulation of technical debt, particularly within the Quality Assurance (QA) and Continuous Integration (CI) processes. As the platform's complexity grew, the initial CI strategies—designed for a smaller, less complex system—began showing their limitations. The need for immediate results often superseded the time required to design optimized, scalable testing architectures.

Consequently, several inefficiencies emerged that now pose a challenge to the wider team's productivity and the infrastructure's scalability. This report focuses on addressing these specific bottlenecks, which serve as the primary motivation for this work. The key issues identified are:

- **Redundancy in Authentication Processes:** Tests that require a repetitive login sequence between file specs and execution instances, significantly increasing the total feedback time.
- **Inefficient Data Management:** The initial testing strategy relies on the excessive creation and deletion of entities, often generating repetitive content that could be reused via templates. Furthermore, this content generation is performed primarily through the Cypress User Interface (UI) rather than faster API mechanisms, leading to slower execution times and increased computational resource consumption.
- **Readability and Standardization:** Most generated test units and templates lack

a standardized process for naming and structuring. This creates friction for external collaborators or newcomers attempting to understand the test suite.

- **Feature Completeness and Alignment:** The pressure for rapid delivery occasionally leads to merge requests reaching the QA phase with missing functionalities compared to original product specifications, necessitating additional iteration cycles.
- **Test Suite Reliability and Maintenance:** To maintain deployment velocity, there has been a tendency to skip failing tests in pipelines rather than addressing root causes—often small UI changes. This is driven by a lack of personnel dedicated to adapting and improving the automated test batch.
- **Excessive Testing Instances:** The current configuration triggers the full test suite on every commit, regardless of the scope of changes, delaying developer feedback and wasting computational resources.
- **Storage Resources Consumption:** The CI pipelines are configured to store full artifacts (recordings and logs) for every run, resulting in excessive storage usage.
- **Task Organization and Handover:** The original workflows were not designed to accommodate the integration of new personnel and larger development teams, leading to unclear task separation.

The motivation for this work, therefore, comes from the critical need to transition from this rapid-growth phase to sustainable, scalable engineering practices. By resolving these issues, the goal is to reduce feedback times, improve system reliability, and empower the development team to maintain high velocity without compromising quality.

1.2 Objectives

Building upon the challenges identified, the primary objective of this internship is to re-engineer the Quality Assurance workflow and infrastructure to support Glartek's scale. The goal is to transition the QA ecosystem from a "delivery-first" mindset to a sustainable engineering practice that guarantees long-term stability and efficiency.

To achieve this, the work focuses on the following specific objectives:

- **Establish a Scalable Quality Foundation:** Transition the testing architecture from a rapid-prototyping model to a standardized, modular system capable of supporting exponential growth in features and team size. This involves enforcing architectural standards that ensure long-term maintainability.
- **Elevate Developer Experience and Efficiency:** Eliminate bottlenecks in the development lifecycle by optimizing pipelines and reducing feedback loops. The goal is to create a lean environment where testing is fast and cost-effective, directly improving the day-to-day experience of the development team.
- **Ensure Reliability and Trust:** Shift the QA culture from reactive maintenance

to proactive quality assurance. By resolving root causes of instability rather than bypassing them, the objective is to restore confidence in the test suite as a reliable safety net for deployment.

- **Streamline Collaboration and Team Growth:** Formalize workflows and knowledge sharing to bridge the gap between development and QA. This aims to facilitate seamless team expansion, ensuring that new members can contribute effectively from day one while reducing the cognitive load on the existing team.

By fulfilling these objectives, this report aims to demonstrate what procedures and techniques a Software Engineer can use to efficiently re-engineer and re-structure a sturdy and scalable testing foundation and architecture, solving technical debt and enabling faster and more secure processes.

2

State of Art

2.1 Continuous Integration and Continuous Delivery

Continuous Integration (CI) and Continuous Delivery (CD) are modern and central topics in the software development field. They describe the processes that automate the execution of different steps in order to have automated builds, continuous testing and seamless deployement, ensuring efficient and clean software product releases and improvements. Some of the key practices include version control integration, automated test execution and deployement orchestration, aimed at reducing manual errors and accelerating delivery cycles [1]. Advanced deployment strategies such as blue/green deployments and canary releases enable zero-downtime rollouts and staged feature validation, while feature flagging provides selective release control and rapid rollback capability. These practices are further strenghtened by Infrastructure as Code (IaC), which enforces environmental consistency across development, staging, and production stages, and by real-time monitoring with fast feedback loops, which together preserve system health and ensure continuous quality visibility throughout the delivery cycle [2].

Recent evidence shows that CI/CD is broadly recognized as a key enabler of faster and more reliable software delivery, but its practical benefits are often constrained by long build times, test selection techniques/challenges and build instability in real-world pipelines [1, 3].

Large-scale evidence literature review [3] also indicate that CI outcomes are multi-dimensional, which reinforces that CI/CD performance should be assessed beyond a single velocity metric. Regarding development activities, CI is strongly associated with improved productivity, efficiency, and developer confidence, although it can also introduce extra complexity and a false sense of confidence when developers blindly trust automated tests that may be flaky or insufficient. From a software process per-

spective, CI acts as a success factor by promoting faster iterations, better transparency, predictability, and stability, while increasing cooperation within teams despite bringing technical challenges related to maintaining build environments and tools. In terms of quality assurance, CI adoption typically leads to an increase in the volume and coverage of automated tests and provides a framework for continuous quality assessment and multi-environment testing. CI also encourages better integration patterns, such as increasing commit frequency and decreasing commit size; while this generally accelerates the pull request lifecycle, some evidence suggests it can occasionally prolong the time to merge due to more rigorous verification. Studies consistently show that CI shortens the time required to find and fix bugs and is associated with an overall reduction in reported defects, and it promotes build practices that lead to better build health and an increase in the build success rate over time.

A critical methodological concern raised by the same review is the inconsistency in how CI adoption is defined across the literature: nearly 42.5% of the primary studies used no explicit criteria, such as commit frequency or automated test requirements, to determine whether a project actually practiced CI before analysing its effects. Most studies simply verified whether a project used a CI service, giving rise to what the authors term "CI Theater", in which projects nominally claim CI adoption but fail to follow essential sub-practices such as immediately fixing broken builds, potentially biasing research conclusions [3].

In parallel, recent research emphasizes contextuality: project characteristics related to product, team, process, quality, and scale significantly affect which practices are transferable, suggesting that pipeline optimization must be context-aware rather than universally prescribed [4, 5]. To address this gap, Huang and Proksch[4] developed a structured taxonomy of CI contextual factors organised into five dimensions: Product, Team, Scale, Process, and Quality. This reinforces the conclusion that there is no one-size-fits-all CI approach; while core principles such as integration frequency stay consistent, implementation pathways varies significantly depending on the specifics of the context. Notably, this study found that the contextual factors most frequently examined in the literature are those easy to mine from repositories, such as the technological environment, project size, and automation levels, whereas human-centred factors like team composition or communication strategies are often overlooked because they are harder to measure at scale. Despite this research-versus-reality gap, the proposed taxonomy received strong practitioner support, with an average agreement rate of 72.9% among surveyed industry professionals, suggesting it accurately reflects real-world CI complexities.

Another key concept to keep in mind is that, with the ever-growing appearance of new tools, technologies and updates of older ones, the CI landscape is in a constant process of change. A study that interviewed 22 participants [5] shows that is common for projects to use multiple CI tools simultaneously in order to broaden en-

vironmental coverage across operating systems and hardware architectures, to access complementary features not available in a single tool, and to manage resource limitations such as build-time restrictions . When projects do migrate between tools, the primary drivers include reducing cost by leveraging free tiers, lowering maintenance burden by moving from self-hosted solutions like Jenkins to cloud-based services, and consolidating pipelines into a single platform. However, the increasing reliance on reusable platform-specific components and the tight integration within social coding platforms introduces a higher risk of vendor lock-in, which teams must weigh against the convenience these ecosystems provide.

Empirical studies on optimization strategies further show that reducing feedback latency is addressed through complementary approaches such as long-running test-suite prioritization [6], predictive CI [7], and workflow-level resource optimization [8], with reported gains in earlier failure exposure, waiting-time reduction, and infrastructure efficiency.

Generally, CI/CD pipeline tend to usually be composed of a small set of recurring steps: source control triggers (commit, merge request, or schedule), dependency setup and environment provisioning, build and static validations, automated test execution at multiple levels, packaging and artifact publication, and controlled deployment to target environments with post-deployment checks and monitoring feedback. In mature workflows, cross-cutting capabilities such as parallelization, quality gates, and failure diagnostics are integrated across these stages to balance release speed with reliability and traceability [5, 8].

2.2 QA fit in CI/CD

Quality Assurance (QA) is more specifically understood as a systematic set of process-oriented activities that ensure software engineering work is monitored against defined standards so that product quality is consistent and trustworthy [9, 3]. In CI/CD environments, QA is often integrated by embedding automated quality checks directly into the pipeline, from build validation and static checks to automated functional, end-to-end, and performance tests. This guarantees each commit is validated continuously rather than only at the end of the development process [10, 5]. Evidence from CI/CD testing strategy studies also shows that distributing different test types across pipeline stages is essential to preserve fast feedback while maintaining broad defect detection in complex systems [11, 6].

2.3 QA Tools

As for the used tools that best implement QA needs in CI/CD, we have Selenium[12], Cypress[13] and Playwright[14] as three of the most frequently used[15] through automated browser-based validation integrated into pipeline jobs. Recent industry anal-

yses also indicate that Playwright is in a rapidly growing trend to overtake both Selenium and Cypress in modern automation stacks [15]. Official documentation for these frameworks highlights CI-oriented execution patterns, including headless runs, reproducible environment setup, and artifact-based debugging support, which aligns their usage with continuous quality gates in pull-request and mainline workflows [9]. In practice, these characteristics make such tools suitable for placing end-to-end and regression checks in stages where fast failure signaling and repeatable diagnostics are required [11, 6].

From a comparative perspective, both frameworks target reliable end-to-end automation, but they make different trade-offs. Playwright provides native multi-browser coverage across Chromium, Firefox, and WebKit, together with actionability-based auto-waiting that reduces timing-related instability in dynamic interfaces [16, 17]. Cypress emphasizes an opinionated developer workflow with built-in retryability and explicit test-retry controls [18, 19], and current evidence from Cypress-focused empirical studies indicates that synchronization and wait-strategy design remain central factors affecting suite stability [20]. For CI/CD design, this suggests a pragmatic selection criterion: teams prioritizing broader browser matrix validation and isolation-heavy execution may benefit from Playwright-first pipelines, whereas teams prioritizing rapid authoring and strong in-tool debugging ergonomics can achieve effective quality gates with Cypress when synchronization practices are well engineered [11].

2.4 QA Challenges and Roadblocks

As CI/CD establish QA more and more as a continuous, automated activity rather than a final step, several challenges persist that can undermine its effectiveness in practice.

2.4.1 Flakiness, Silent Failures and Developer Concerns

Flaky and intermittent job failures have emerged as a major obstacle to reliable CI/CD, not only because they waste machine resources through repeated reruns, but also because they are rooted in heterogeneous causes that extend beyond the source code itself, including unstable tests, transient network faults, authentication problems, and infrastructure instability. This heterogeneity makes diagnosis particularly difficult and suggests that such failures should not be treated as a single uniform phenomenon. In this regard, Aïdasso et al. analysed 4,511 flaky job failures and showed that 40% of them fell into only five categories, with misconfigured environment variables alone accounting for 15% of all occurrences. Their study further identified 14 priority categories by combining recency, frequency, and cost, indicating that diagnosis and repair efforts should be directed first toward the most recurrent and operationally expensive failure classes rather than applied uniformly across all flaky events [21].

Within this broader space, intermittent job failures represent a particularly costly subset because they are non-deterministic and therefore frequently misinterpreted by existing detection methods. In an empirical study covering 16,786 failed jobs across five industrial projects and one open-source project, the same researchers found that the state-of-the-art automated labeling strategy based on non-deterministic reruns misclassified, on average, 31.97% of failures as regular when they were in fact intermittent, with project-level error rates ranging from 16.18% to 50.14% [22]. This result is methodologically significant because it shows that rerun-based heuristics depend too strongly on local engineering practices: when suspicious failures are not systematically rerun, the resulting labels become unreliable and can distort downstream machine-learning models. To address this limitation, the authors proposed a few-shot learning framework based on a fine-tuned small language model and only 12 manually labeled examples per class, achieving F1-scores between 70% and 88% across all studied projects and outperforming the prior state of the art in five of the six cases, where baseline scores dropped to 34–52% in the projects with the highest mislabeling rates. Although cross-project prediction remained less reliable than project-specific models, it still surpassed the baseline in four of the six projects, suggesting that language-model-based log understanding offers a more robust mechanism for triaging intermittent failures while reducing wasted reruns and diagnosis effort. This broader direction is also reinforced by Moriconi's [23] doctoral work, which shows that data-driven analysis of CI/CD logs can support automated root-cause analysis beyond simple failure detection, including build-failure classification with 94% accuracy and root-cause message localization through recurrent neural models without prior root-cause annotations, thereby strengthening the case for richer log-based support in unstable pipelines [22].

A further, less visible reliability threat is the occurrence of silent failures, in which jobs are marked as successful despite failing to complete all their intended tasks. In the first empirical study of this phenomenon, Aïdasso et al. [24] analysed 142,387 jobs across 81 industrial projects and found that 11% of successful jobs had been rerun, accounting for 51% of all reruns without code changes, compared with 44% for failed jobs. More than 35% of these reruns were triggered only after 24 hours, sometimes after more than 72 hours, and they consumed 48% of the total server time spent on all job reruns, showing that silent issues are both prevalent and costly . The same study also identified the circumstances and technical patterns most associated with these hidden failures: successful-job reruns were more likely in testing and static-analysis jobs, in scripting-heavy environments such as shell, and in projects with larger development histories, while the most common silently failing tasks involved artifact operations (28%), caching errors (23%), and ignored non-zero exit codes (18%). Taken together, these findings show that "green" builds cannot always be interpreted as trustworthy success signals, and that improving CI reliability requires explicit attention not only to failed jobs, but also to successful executions that may conceal incomplete or incorrect outcomes.

Taken together, these studies suggest that the most important consequence of flaky, intermittent, and silent failures is the erosion of developers' trust in QA and CI feedback. The systematic review by Soares et al. [3] highlights that CI effectiveness becomes particularly complex in settings involving emulated environments, non-deterministic tests, and inconsistent execution environments, precisely because such contexts weaken the reliability of automated feedback . The same review also reports a risk of a false sense of confidence, defined as situations in which developers blindly trust tests despite the underlying environment suffering from low-quality or insufficient test suites. Although the evidence for some negative effects is methodologically weaker and requires further empirical confirmation, the practical implication is clear: trust in CI should not be treated as an automatic by-product of automation, but as something that must be earned through trustworthy environments, stable tests, and objective criteria for assessing whether pipeline feedback is genuinely reliable.

2.4.2 Test Redundancy and Test-Suite Optimisation

Beyond reliability concerns, test redundancy and suite bloat represent a growing cost problem in CI/CD. As projects evolve, test suites accumulate cases that overlap in the faults they detect, inflating execution times without proportional gains in defect coverage. This has motivated a wide range of regression-test optimisation techniques, including Regression Test Selection (RTS), Test Case Prioritisation (TCP), and failure-prediction approaches that seek to execute only the tests most relevant to a given change. These fine-grained strategies are particularly valuable for large and frequently triggered pipelines because they reduce waiting time and computational cost while preserving high fault-detection capability [6, 25].

A representative large-scale example is Google's transition-based test selection (TBTS) approach. As reported by Kodithyala [25], TBTS was evaluated over more than 2.5 million builds and 50 million test executions, reducing the number of executed tests by up to 95% while preserving 99.6% fault-detection capability. For typical code changes, execution time decreased from 35 minutes to 3 minutes, corresponding to an 87% reduction in code integration time. More broadly, recent AI-driven approaches extend this logic by using historical execution data and predictive analytics to forecast build outcomes and prioritise failure-prone changes [7, 26]. They can also support diagnosis from complex build traces, thereby reinforcing test selection as both a quality-preserving and resource-aware optimisation strategy [2].

This focus on regression-test optimisation is further reinforced by workflow-level evidence on CI resource consumption. Bouzenia and Pradel[8] show that testing and building account for 91.2% of the resources consumed in GitHub Actions workflows and estimate an average annual cost of approximately $504 for a paid-tier repository, which helps explain why regression testing has become such a critical optimisation target . Their study also highlights that fine-grained techniques such as test selection,

test prioritisation, failure prediction, and commit prioritisation are complementary to broader workflow-level optimisations rather than replacements for them. In particular, mechanisms such as caching and timeout configuration remain underused despite measurable impact, suggesting that effective test-suite optimisation in CI/CD should combine test-level reduction techniques with workflow-level resource management, since both address different but complementary sources of inefficiency. Complementarily, many of these cost pressures can also be mitigated earlier in the lifecycle by shifting validation activities to the left, before defects propagate into later and more expensive stages.

2.5 Shift-Left Paradigm and Cost Efficiency

The cost dimension further underscores the importance of early detection. In CI/CD contexts, the "Shift-Left" paradigm promotes moving testing and security checks to earlier lifecycle stages so that defects can be identified and addressed while feedback is still immediate and corrective action remains comparatively inexpensive. This rationale is reinforced by evidence that fast feedback loops are central to CI/CD effectiveness and by data-driven studies showing that the financial impact of defects escalates sharply as they progress through the software lifecycle: Moriconi [23] reports that fixing a bug during testing can be around 15 times more expensive than during design, and up to 100 times more expensive if deferred to maintenance [1]. In DevSecOps settings, this logic extends directly to security validation through Continuous Security Testing, which integrates security checks earlier rather than postponing them to later release stages, thereby aligning security assurance with the same early-feedback and cost-mitigation principles that justify shift-left testing more broadly [27].

From this perspective, continuous testing becomes the operational expression of shift-left in CI/CD. Rather than treating QA as a final gate, continuous testing distributes validation throughout the software delivery process, combining automated and manual checks to repeatedly assess quality as development progresses [28]. In practice, this means integrating multiple test types into the pipeline, including unit, integration, regression, and system-level checks, together with non-functional validation such as performance, reliability, and security testing. Build-oriented studies similarly describe CI pipelines as automated sequences that routinely combine static analysis, compilation, unit testing, and deployment-related tasks to ensure the quality of integrated changes and the reliable production of new software versions [10]. Taken together, these findings reinforce that modern QA in CI/CD is not a late-stage verification activity, but an ongoing and multi-layered process performed in parallel with development.

2.6 Lean Software Development

Lean thinking originated in the Toyota Production System (TPS), as a manufacturing philosophy developed to maximise value delivery by systematically identifying and eliminating waste, defined as any activity that consumes resources without contributing value to the end product [29]. When the Poppendieck translated these principles into a software engineering context [30], they defined seven categories of software waste that mirror the original manufacturing concept: partially done work, extra features, relearning, handoffs, task switching, delays, and defects. Each category represents a form of effort that reduces the throughput and reliability of the delivery process without improving the final product, providing the software industry with a vocabulary for diagnosing inefficiency at the process level rather than the individual task level.

In practice, Lean Software Development proposes seven guiding principles: eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimise the whole [30]. Several of these are particularly relevant to QA process design. Building quality in, rather than inspecting it at the end, directly motivates the shift-left strategies discussed in the previous section, where defect detection is moved to earlier, cheaper lifecycle stages. Delivering fast maps to the CI/CD objective of minimising feedback latency and integration cycle times. Optimising the whole counters the tendency to improve individual pipeline stages in isolation, which can shift rather than eliminate bottlenecks. Together, these principles provide a coherent rationale for re-engineering QA workflows: the goal is not to add more checks, but to reduce the friction, delays, and redundant effort that prevent existing checks from delivering value predictably.

Applied to software process re-engineering, the Lean perspective establishes that a structured diagnosis of where waste accumulates in the current workflow should precede any solution. In QA contexts, common sources of waste include long manual verification cycles that delay feedback, test suites with redundant or low-value cases that consume pipeline resources without proportional defect detection, unclear task handover processes that cause avoidable rework, and fragile test data setups that introduce non-deterministic failures. Addressing these systematically — rather than patching individual symptoms in isolation — is what distinguishes process re-engineering from localised debugging, and it is the guiding rationale for the intervention described in the following chapter.

3

Process Re-Engineering

This chapter presents a high-level overview of the re-engineering work carried out during the internship. Rather than focusing on implementation details, it summarizes the main problems observed in the existing QA and CI/CD flow, the reasoning behind the selected interventions, and the areas in which the new approach was expected to improve efficiency, stability, and maintainability. Detailed implementation choices and concrete technical examples are presented in the following chapter.

When this work started, the validation process combined a significant amount of manual verification with a growing automated end-to-end test battery. Although this structure already provided quality control, it also introduced long feedback cycles, duplicated effort between team members, and unnecessary consumption of machine and runner resources. This causes, joined with a limited ammount of manpower caused the QA department to be oftentimes the bottleneck of the development cycle. The re-engineering effort present here, informed by Lean Software Development principles [30], is focused on maintaining the core validation goals while reducing waste and improving the predictability of the overall delivery process.

To make the intervention easier to understand, this chapter is divided into four major areas of impact: workflow, software architecture, data and test restructuring, and tooling and migration support. Together, these areas represent the main pieces of the process that were adjusted in order to make QA more scalable and better aligned with CI/CD principles.

3.1 Workflow

The first area of change concerned the way work reached QA and how responsibilities were distributed across the delivery cycle. One of the main observations was that QA was frequently treated as the point where unfinished or weakly validated work was completed, instead of being used as a final quality checkpoint for work that was already

technically ready for deployment. This behaviour created avoidable iteration loops, delayed merge requests, and reduced the time available for higher-value validation activities.

To address this, the workflow was redefined to clarify task ownership and expectations before and after a testing request reached QA. Changes were introduced so that developers assumed more responsibility for the delivery quality of their own features, which helped detect obvious issues earlier and reduced the number of avoidable back-and-forth cycles. A general view of this development cycle is presented in Figure 3.1. The boxes highlighted in blue present the steps of the process that were investigated and re-engineered the most, as well where participation was most clear and impactful, contributing to the results presented in this document.

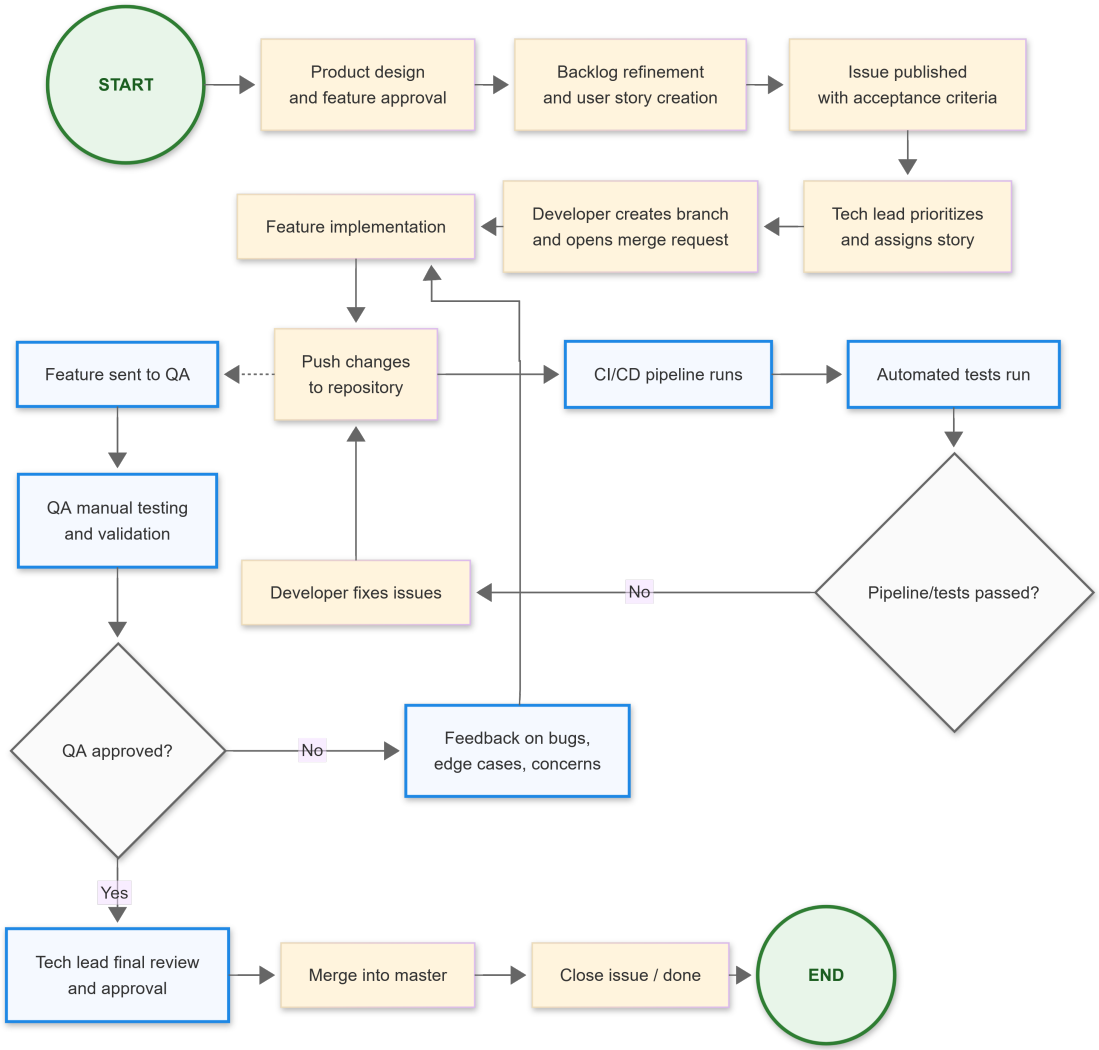

Figure 3.1: *High-level feature approval workflow of product, development, CI, and QA interactions.*

The process also benefited from the introduction of clearer organisational signals inside the development flow. Labels and lightweight guidance mechanisms helped distinguish development stages and distribute tasks among QA team members more

transparently. As a result, traceability improved and ownership became less ambiguous throughout the validation process.

Documentation refinement also became a core principle of the redesigned structure. Clearer and more up-to-date documentation was treated as a cornerstone for handing down knowledge, reducing dependence on informal explanations, and stabilising work procedures over time. In practice, this helped preserve process consistency, made onboarding and task handover easier, and supported a more reliable execution of recurring QA activities.

From a reporting perspective, these workflow changes are important not because they introduce any new tools, but because they reshape the behaviour around the existing toolchain. In practical terms, the objective was to shift QA effort away from repetitive coordination work and toward product validation, exploratory analysis, and the identification of issues that are harder to detect through purely automated means.

3.2 Software & Hardware Architecture

At an architectural level, the re-engineering work aimed to simplify both the codebase and the development flow by removing, relocating, or reducing steps whose contribution to QA was limited. Some auxiliary actions in the process generated artifacts or intermediate operations that were not essential for decision-making, increasing execution time and resource consumption without creating corresponding value. Reducing these elements became one of the first opportunities for immediate improvement.

Another relevant discussion concerned the environment in which tests should run, namely whether the preferred approach should rely on locally prepared environments or on shared development deployments. An evaluation period was used to assess which architectural direction offered the best balance between realism, reliability, and resource usage. In parallel, the order of execution of tasks inside QA validation flows was also reconsidered in order to speed up feedback and reduce avoidable infrastructure costs. This execution model is illustrated in Figure 3.2. As for the directly and most affected area in this document, the almost totality of changes in this pipeline connected to the work present here are inside the ”Runner with Provisioned Environment” block.

3.3 Data and Test Restructuring

A substantial part of the re-engineering effort focused on the structure of the automated test suite itself and how it is integrated within the product. Over time, the existing tests had accumulated hardcoded values, inter-test dependencies, and data management patterns that made maintenance harder and reduced confidence in the stability of the suite. The goal was not only to fix isolated failing tests, but to reorganise the suite so that new tests could be created, located, and maintained more consistently. This

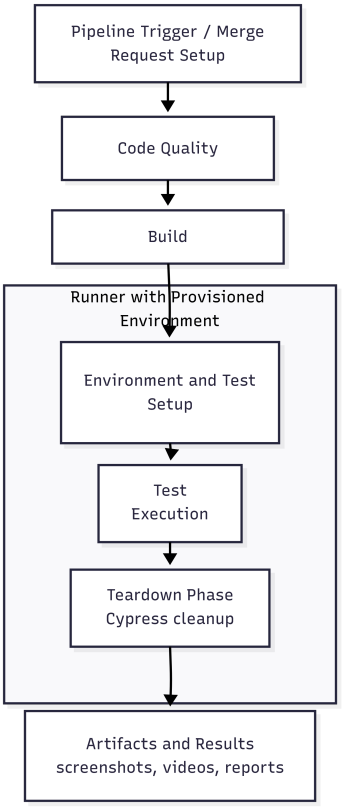

Figure 3.2: *Cypress-based testing pipeline executed against runner-provisioned environments.*

restructuring also created the conditions for introducing new test categories — API integration tests and component-level unit tests — to progressively distribute coverage across a more balanced testing pyramid, reducing dependence on the most resource-intensive end-to-end layer.

One of the most important improvements in this area was the standardisation of variables and naming conventions shared by tests and seed data. By reducing scattered definitions and intermediate naming layers, the suite became easier to navigate and less dependent on implicit knowledge. This simplification also made it easier to update existing scenarios and to reason about the relationship between test declarations and the data required to execute them.

Another key change was the effort to isolate tests more clearly. Separating specifications so that each file focused on a single test case or a tightly scoped behaviour improves readability, reduces accidental coupling, and supports maintenance over time. Organising these tests by functional area also makes the suite easier to search and reason about, especially as the number of scenarios increases.

The same restructuring logic applied to test data management. Instead of relying excessively on manually created data or on heavy cleanup routines after execution, part of the effort involved moving reusable scenarios into seed-driven data preparation. This reduced redundant setup work, shortened test bodies, and decreased flakiness

caused by side effects between scenarios.

The seed-data layer itself also required improvement, since it had become difficult to maintain and reason about. Part of the existing logic was spread across three files and combined hardcoded queries with executable helper functions, which made the structure harder to follow and increased the effort required to update or debug test data preparation. Refining this area was therefore an important step toward making the overall testing workflow more consistent and maintainable.

A structured analysis was also conducted to determine which existing end-to-end tests warranted adaptation to the new structure and which should instead be migrated to API-level tests, establishing the migration order before any investment was made in adapting tests that were subsequently to be removed from the E2E battery.

In addition, parallel execution of specifications became a more relevant topic as suite growth imposed increasingly long waiting times for test runs, which indirectly increased integration time and delayed feedback to developers.

Finally, it was decided to move the end-to-end tests outside the UI folders in order to treat them as an independent testing resource rather than as a tightly coupled part of the UI codebase.

3.4 Tools and Migrations

The re-engineering process also involved evaluating the tools that supported local development, automated execution, and future maintenance of the QA ecosystem. Beyond improving the current test battery, it became necessary to assess whether the existing stack still represented the most adequate long-term option for the team, particularly from the perspectives of reliability, debugging, and execution speed.

In this context, Playwright emerged as an important candidate for experimentation and possible adoption in future iterations. The purpose of this evaluation was not to replace the current stack impulsively, but to understand whether a different framework could offer practical advantages in terms of test isolation, execution model, maintainability, or developer experience. This exploratory work therefore acted as both a technical validation exercise and a strategic preparation for future migration decisions.

Supporting scripts creation were also relevant in reducing friction in the local workflow. Automating the startup and seeding of the environment helps create reproducible execution conditions and lowers the time required to prepare a testing session. A script capable of rebuilding the local state, cleaning previous data, and reinitialising the required services is especially useful in teams where environment drift can quickly become a source of false failures and debugging overhead.

Additionally, a report and execution-log analysis played an important role in identifying unstable tests and prioritising correction work. Rather than treating all failures

equally, the goal was to understand which parts of the suite showed the highest instability, why those failures occurred, and which corrections would generate the highest impact. This diagnostic perspective is essential in re-engineering work because the objective is not merely to grow the number of tests, but to increase the trustworthiness and operational value of the suite as a whole.

Finally, a dedicated feedback recording tool was developed to reduce friction in the process of reporting bugs and improvement suggestions. By integrating screen capture directly into the team's existing GitLab-based workflow, this tool removed the dependency on external recording applications and standardised the format in which feedback reached the development team.

3.5 Lean Principles in Practice

The four areas described in this chapter were not defined independently. Each one was shaped, at least in part, by the Lean principles introduced in the previous chapter, and understanding that connection helps explain why certain interventions were prioritised over others.

The most directly visible principle is *eliminate waste*. Redundant teardown blocks, fragile test cases, and avoidable coordination steps were removed wherever they appeared, particularly across the Workflow and Data and Test Restructuring areas. Closely related is *build quality in*, which informed the shift-left strategy and the move toward seed-driven data preparation: the goal was to prevent environment-induced failures structurally rather than catch them late. *Create knowledge* shaped the documentation and naming convention work, which was treated not as administrative overhead but as a prerequisite for stable handover and consistent execution.

Deliver fast guided the automation of local environment setup and the pipeline changes aimed at shortening feedback cycles. The principle of *optimise the whole* was perhaps the most important constraint throughout: it pushed back against changes that looked efficient in isolation but would have shifted problems elsewhere in the system, keeping test structure, tooling, pipeline, and workflow changes aligned.

The two remaining principles, *defer commitment* and *respect people*, were present in a less prominent way. Decisions with long-term consequences, such as whether to migrate to Playwright, were deliberately kept open until enough evidence had been gathered to justify them. Clearer ownership signals, reduced blocking handoffs, and documentation written to stand on its own without informal context all reflect the human side of the re-engineering work.

Lean was useful here not as a rigid framework but as a shared vocabulary for reasoning about trade-offs. The following chapter details the concrete implementation behind each of these areas.

4

Implementation

This chapter describes the decisions that led to changes, refactors, and restructuring in QA, as well as their impact on the broader development lifecycle. As discussed in the previous chapter, these changes did not come from a single coordinated intervention, but from an iterative process shaped by the constraints and particularities of the context. Different solutions had to be proposed, tested, and reviewed to understand whether they aligned with the company's internal guidelines and objectives. Throughout this process, one condition remained constant: changes could not disrupt the normal functioning of QA.

In practice, this iterative process was mainly carried out through meetings focused on improving development lifecycle procedures. These meetings involved the intern and a group of experienced developers, including the lead representative of the R&D department and another senior contributor with strong alignment with the company's strategic direction, so that the discussed topics could be evaluated against both roadmap and business priorities. They took place at intervals ranging from one week to one month, depending on the estimated complexity and effort of the proposed changes. This rhythm allowed interventions to be implemented step by step and then reviewed according to their observed effects, making it possible to revise or postpone decisions whenever the expected benefits were not achieved or when further analysis was still needed.

4.1 Workflow

In this section, we describe the changes made to workflow procedures and to the way tasks were handled. The focus is therefore more behavioural and organisational than technical.

To better contextualise these scenarios, it is important to note that the company

went through a renewal of its QA workforce. A junior developer team joined the company and became the foundation and support of the QA department, supervised by two more experienced developers who moved from QA to development when the junior team joined. This created a context in which existing methods could be reworked and redefined with less internal friction, since the new team members were not transitioning from an old workflow, but instead directly implementing the newly proposed one.

4.1.1 Task Separation and Handover - Labels and QA

These initial adjustments consist of smaller changes with impact, but without major effects outside QA boundaries. Their value is mostly intrinsic, improving the developer experience and efficiency of the QA team.

Given the context described above, one of the first priorities was to access and distribute the existing QA workload among the new team members through a clear, structured process.

The main existing task, and the most crucial for the QA department, is manual testing. It is a continuous process in which the QA team receives Merge Requests (MRs) labelled for testing and then starts validation. This step is represented by the "Feature sent to QA" box in Figure 3.1. To support this, QA members should actively keep GitLab open and filter existing MRs by the label "Testing required", while excluding MRs assigned to other QA members. This is a continuous and spontaneous activity that should happen whenever a QA engineer finishes an ongoing task. The expected work listing is illustrated in Figure 4.1.

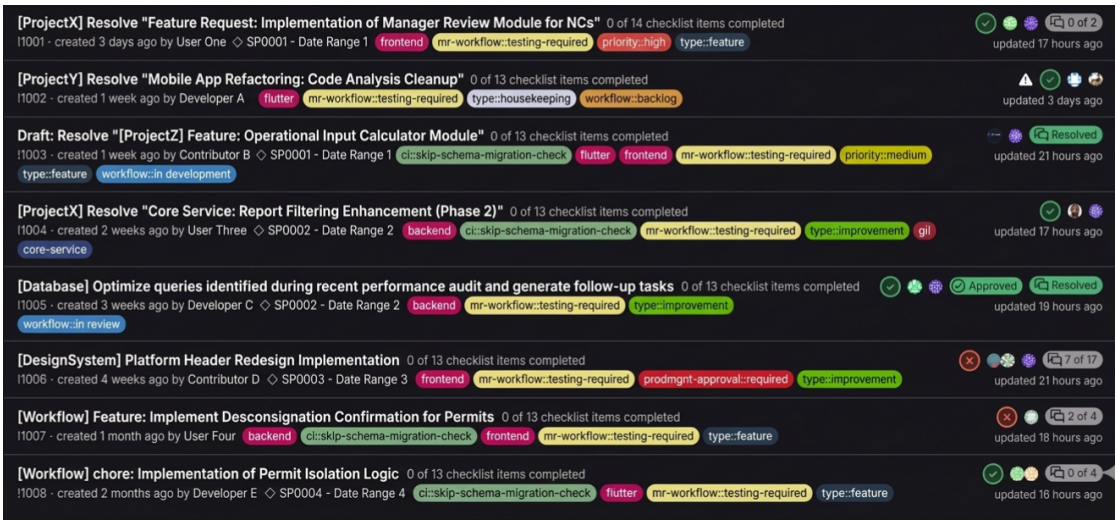

Figure 4.1: *Merge requests listed after filtering by the* `Testing required` *label, representing the typical manual testing queue.*

Once the work queue is identified, a QA developer selects the highest-priority MR and assigns themselves to it. This assignment was one of the implemented changes and

introduced a clearer mechanism for communicating ownership of testing tasks within the team. More specifically, it provided three main benefits:

1. It prevented two QA members from starting work on the same MR simultaneously, thereby avoiding duplicated effort.
2. It improved work navigation and filtering, as MRs without a QA assignee could be easily identified as still awaiting manual testing.
3. It made the testing status of each MR more visible to both QA members and developers, since the assignment of a QA member corresponded to the first step of the manual testing process represented in Figure 3.1 by the box "QA manual testing and validation".

This also made labels one of the main mechanisms supporting task distribution. At the beginning of the re-engineering process, the QA workflow relied on three dedicated labels:

1. Testing required
2. Testing successful
3. Needs automated testing

By the end of the process, three additional labels had been introduced to cover scenarios that were not adequately represented by the initial set. The newly added labels, each created for a specific purpose, were the following:

1. qa manually tested - manually looks good
2. qa manual testing skipped
3. automated tests - needs improvement

Among these, `qa manually tested` indicates that the MR has been manually approved by the responsible QA member and is only waiting for the automated test suite to pass. In turn, `qa manual testing skipped` is used for MRs involving minor bug fixes. This label became necessary as the number of MRs increased and a decision was made to transfer testing responsibility for minor bug-fix MRs to developers.

Finally, due to the increasing size and complexity of the automated testing battery, the label "automated tests - needs improvement" was introduced to associate unstable automated tests with the corresponding issues on the GitLab platform. This made it possible to identify which closed issues had generated automated tests that were still unstable and required further refinement. For the QA team, the main benefit was practical: whenever a test failed in the automated testing pipeline, the responsible QA developer could consult the issue associated with that test and, upon seeing this label, quickly recognise that the failure involved an already known unstable test requiring improvement.

4.1.2 Responsibility Ownership

A recurring topic within the development lifecycle was the delay caused by repeated iterations between developers and QA. As represented in Figure 3.1, this loop begins during feature implementation and continues until QA approves the MR after validating that no manual defects, relevant edge cases, or other concerns remain. In practice, this created a grey area in the process, caused by the difficulty of defining where developer-side testing should end and where QA validation should begin.

A common example illustrates this problem clearly. After implementing a feature and validating only the main workflow, a developer could send the MR to QA and wait for manual testing. However, QA review was usually not immediate, as the responsible QA member might still be occupied with other MRs. This created waiting time that could range from a few hours to several days, depending on feature urgency and the current QA workload. This effect can be observed in Figure 4.1, where the time shown on the right side of each MR reflects the time since the item entered the manual testing queue.

The cost of this delay extended beyond waiting time alone. Once a QA member picked up the MR, they first had to regain context about the feature and define MR-specific validation paths and edge cases. Although part of this effort could be reused in future iterations, it still depended heavily on the QA member's memory and on their ability to track what had already been tested and what still needed to be verified if the MR returned to development. When several MRs accumulated at the same time, this also generated confusion and increased the feeling of overload within the QA team.

Impact on the Delivery Cycle

Because this was an iterative loop, the inefficiencies described above were not isolated events but repeated sources of friction. Each additional cycle increased internal frustration, delayed MRs, and consumed time that could otherwise have been used for higher-value validation work. For this reason, one of the central objectives became reducing the number of QA–developer iterations as much as possible, ideally reaching a state in which QA would need to manually validate an MR only once before approval.

Reducing these iterations also had an important secondary effect: it lowered the amount of time spent by QA on repetitive manual validation, creating more room to improve the automated testing battery. This was important because stronger automated coverage indirectly reduces future manual effort by increasing confidence in MRs that have already passed the pipeline, particularly with regard to regressions and implementation mistakes introduced while reusing or adapting existing platform components.

However, reducing this cycle required drawing a clearer boundary: features sent to QA should already be fully implemented and thoroughly tested by the developer.

Although this shifts part of the testing burden toward developers and may slightly extend the feature development stage, it does so more efficiently than relying on QA to absorb that responsibility later in the process.

This position was supported by three main reasons. First, developers generally have a better understanding of the features they implement and can more easily navigate the conditional paths and corner cases that often lead to rework. Second, while the feature is still under development, testing can be performed continuously as implementation progresses, allowing defects to be detected closer to their source. By contrast, when QA receives a completed MR that may span several pages or behaviours of the application, the QA member must first reconstruct the feature as a whole, which increases the effort required to identify all relevant validation paths. Third, reducing iterations also reduces the "dead time" during which an MR is waiting for manual testing and no one is actively working on it.

It is also important to note that features were often sent to QA without sufficient prior validation mainly due to time pressure and workload. Under high delivery pressure and strong client-driven urgency, there was a tendency to move implementation forward as quickly as possible and implicitly transfer a substantial part of the testing effort to QA, assuming that the MR was likely already ready for deployment.

Proposed Solutions

This discussion raised the broader question of freedom-based versus control-based policies. Some proposals suggested limiting the number of times the same MR could be sent to testing and applying penalties to developers who exceeded that limit, such as lowering the priority of their work in the QA queue. Another proposal suggested a negative-scoring leaderboard in which developers would accumulate points whenever one of their MRs was returned to development. Both approaches were rejected because they conflicted with the company's teamwork principles and were likely to introduce unnecessary stress and pressure.

The alternatives considered more aligned with the intended objectives were lightweight guidance mechanisms rather than punitive controls. One proposal consisted of introducing a confirmation modal when developers changed the MR label to "Testing required". The modal would ask them to explicitly confirm that the feature had been implemented and tested and should be ready to pass QA validation and be integrated into the platform. The purpose of this mechanism was to reinforce responsibility ownership by reminding developers that they were accountable for the quality of the delivered feature and that QA should act as a final validation stage rather than as support for unfinished implementation work.

This proposal was particularly relevant because QA represented only a small fraction of the total development team, approximately 15%, and therefore could not realistically function as implementation support for the entire engineering organisation.

The proposed modal is illustrated in Figure 4.2. Although the idea was well received in the meetings in which it was discussed, it had not yet been implemented by the end of this report due to higher-priority infrastructure migration work and the resource constraints of that period.

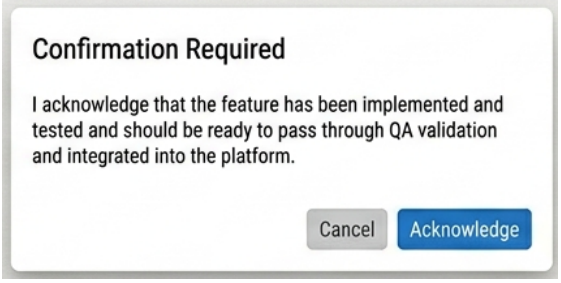

Figure 4.2: *Mock-up of the proposed confirmation modal when setting the* `Testing required` *label. Image generated with Google Gemini.*

Another aligned proposal was to rename the label "Testing required" to "Tested by developer". Although the two names refer to the same stage in the workflow, the second formulation places clearer emphasis on the expectation that QA validation should only begin after development-side testing is complete. As with the confirmation modal, the objective of this proposal was to strengthen the perception of developer responsibility and reduce avoidable iterations in the developer–QA cycle.

Implemented Proposal

One solution was agreed upon and effectively implemented across the development lifecycle: the reviewer role gained substantially greater relevance. Previously, this role was used mainly to associate a more experienced developer with a junior developer whenever additional support was needed. In the revised workflow, however, the reviewer became a more active participant in the validation of the implemented code, with the responsibility of analysing the solution in greater depth and identifying aspects that required closer attention or restructuring.

This change introduced an earlier review layer before the MR reached QA, improving implementation quality and reducing the likelihood that obvious issues would only be detected during manual validation. In practical terms, it also helped prevent technical debt from being introduced unnoticed, since weaknesses in the implementation could be challenged at an earlier stage by a more experienced developer.

These changes also prompted the definition of responsibilities for the new types of tests that were introduced as part of the broader restructuring effort described later in this chapter. Since these test types did not previously exist within the company, their ownership had to be established from the ground up. Table 4.1 summarises how those responsibilities were distributed across roles for each test type.

Table 4.1: *Responsibility matrix across test types in the revised development lifecycle.*

Who?	UI Component	Functional (API)	E2E
Creates Acceptance Criteria	Product & Design	Product & Tech Lead	Product & Tech Lead
Develops the Tests	MR Dev	MR Dev	QA
Required Reviewer	Tech Lead	Tech Lead	Tech Lead
Domain Reviewer	Design	Product	Product

Workflow Shift and New Development Cycle

Taken together, the changes discussed and implemented throughout this report resulted in a broader shift in the development lifecycle. As will be further discussed in the sections on Software Architecture, Data and Test Restructuring, and Tools and Migrations, the developer–QA loop was pushed to a later stage of the process, taking place only after the pipeline had completed successfully.

This shift was made possible by the strengthening of the automated testing battery and by the clearer redistribution of responsibilities across the lifecycle. As argued earlier, a more structured and stable battery reduced the number of iterations between development and QA by increasing confidence in the state of the MR before manual validation. In turn, QA could perform approval with less friction and devote more attention to exploratory analysis, edge cases, and details that might otherwise go unnoticed under the mental burden imposed by repeated corrective cycles.

This change also implied a stronger expectation that developers would be responsible for resolving any pipeline failures introduced during feature development. Confidence in the battery therefore became an important enabling factor: once the automated suite was considered sufficiently reliable, passing pipeline validation could be treated as a prerequisite for manual QA rather than as a parallel or later concern. Altogether, these changes led to a revised workflow, illustrated in Figure 4.3.

4.1.3 Documentation

Documentation refinement was another relevant part of the implementation effort, perhaps the most impactful goal in a longer time-frame perspective. The goal was not simply to produce more documentation, but to make it more concise, usable, and aligned with the actual working procedures adopted by the team. In practice, this involved reviewing outdated material, reducing the visibility of content that was no longer useful, and prioritising the most operationally relevant guidance.

QA Rulebook

The first step of this work was rewriting the rulebook. Although a rulebook already existed, its extended length and the need for regular updates as the test infrastructure and platform grew had caused it to go largely unnoticed, functioning primarily as a supplementary reference rather than a central guide. The goal was to reposition this

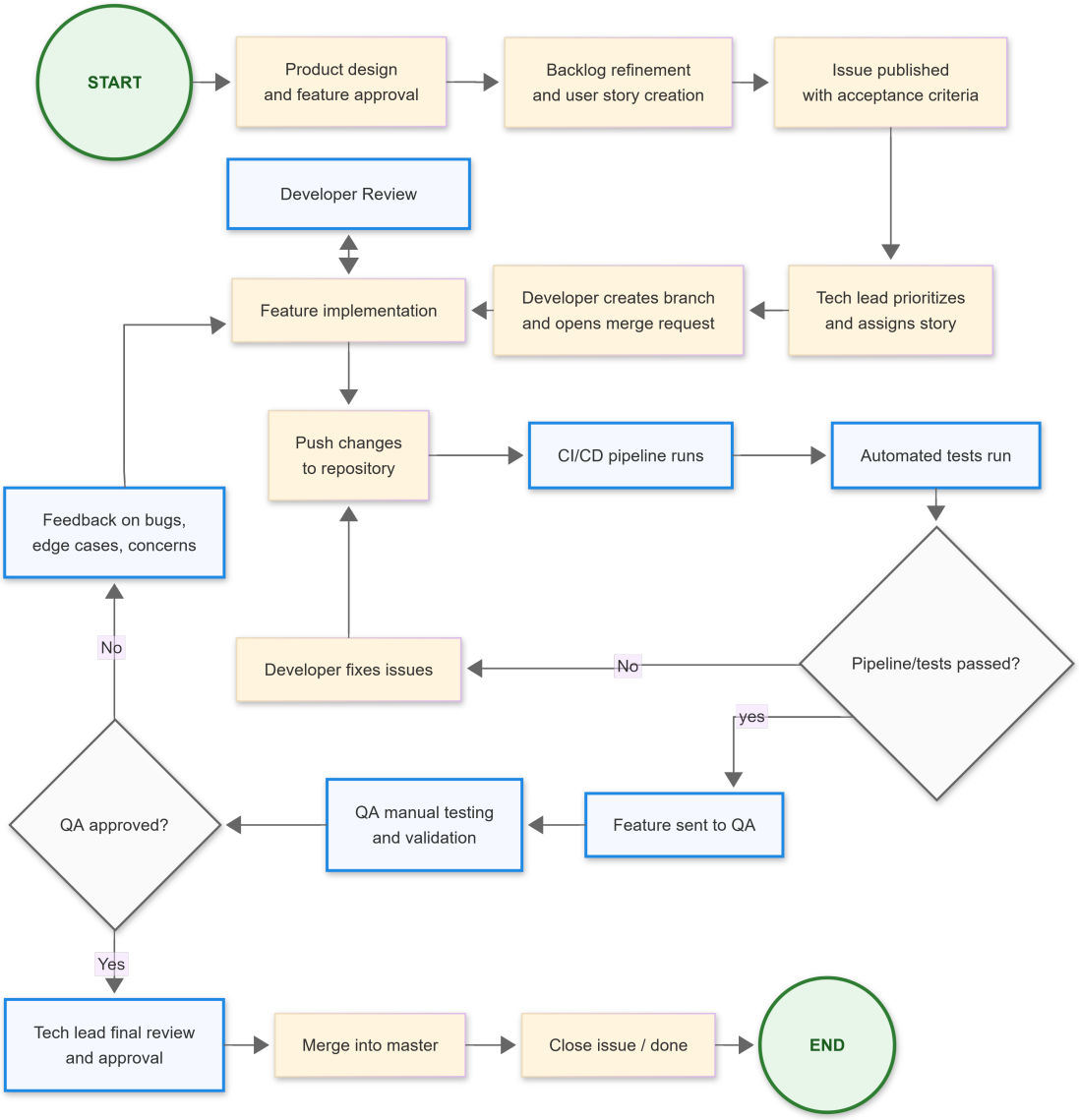

Figure 4.3: *Revised feature approval workflow after the implemented process changes.*

document as the primary QA reference by making it more concise and accessible, so that not only QA members could consistently follow the defined procedures, but also members from other areas of the company could gain a clear overview of the established processes, thereby strengthening the alignment between QA, development, and product design.

Testing Framework Guides

The introduction of new test types and the broader restructuring of the test suite described in the Data and Test Restructuring section created a corresponding documentation need. As new QA team members joined and developer responsibility for certain test categories was formally established through the distribution described in Table 4.1, the absence of centralised, structured guidance on how the framework was organised and how tests should be created and maintained became a gap that needed to be addressed.

The first piece of documentation produced for this topic was a framework overview. Containing the table already presented in the Workflow section, this guide built on it by clarifying the purpose of each test type, the conditions that should lead to the creation of a new test, the location of each test category within the repository, and who held ownership over its development and review. It also described the structural relationship between the three layers of the testing pyramid and how the balance between E2E, API integration, and component tests was expected to evolve as the suite grew, consistent with the direction outlined in the New Types of Tests section. The goal was to provide a single reference point that any developer or QA member encountering the test suite for the first time could consult to understand not just where things were, but why they had been organised that way.

Alongside this, a set of structured improvement guides was produced to support the ongoing process of expanding and refining the battery. These documents described the improvement cycle as a sequence of discrete, ordered tasks: from identifying candidates for migration down the testing pyramid to executing the necessary structural changes, so that the work could be picked up by any team member without requiring prior detailed knowledge of the full process. This was particularly relevant given that the battery was actively evolving, with multiple types of changes taking place simultaneously across seed data, spec structure, and test placement, as described throughout this chapter.

A further set of guides addressed the initial environment setup, the test data structuring process, and the overall mechanics of assembling a new test. These documents combined text with diagrams to make explicit steps that were otherwise implicit: how seed data was organised, how a test spec should reference it, and how the environment was provisioned before a run. Their primary audience was new QA members joining the company, or existing company members seeking a clearer understanding

of QA procedures and infrastructure. The intent was to ensure that the structural decisions described in the Seed Data Restructuring and Test Structure and Conventions sections would be consistently applied by anyone adding to the battery, rather than being known only to those who had been involved in the original restructuring effort.

Finally, this phase of the process also involved pruning existing content. Pages and components covering processes that had since been replaced or were no longer operationally relevant, such as the previously used external tool for injecting seed data into the backend, which had since been replaced by the restructured seed approach, were archived or made less prominent. This focus on retaining only what was actionable and aligned with the current working approach contributed to documentation that was easier to maintain and more likely to function as a reliable reference for the team.

Tool Usage Guides

The adoption of new tooling, particularly the migration of the E2E battery to Playwright described later in the Tools and Migrations section, created a parallel documentation need. The documentation included tips on day-to-day usage of the framework, including writing, running, and debugging tests, to extend knowledge across the QA team and, for certain test categories, to feature developers as well. Without a centralised reference, this knowledge would have remained anchored to the few members directly involved in the migration, limiting the ability of the rest of the team to contribute to the battery without dedicated assistance.

To achieve this, a set of guides was produced covering the practical aspects of working with the new framework. These documents described the initial environment setup, the configuration steps required to run tests locally, and the most common usage patterns expected during test development and debugging. The goal was to lower the barrier to entry for any team member encountering the new framework for the first time and to ensure that the conventions established during the migration were consistently followed by anyone subsequently contributing to the battery.

Alongside these, a complementary set of notes was produced on the use of MCP servers[1] within the development process. The integration of MCP-based tooling represented a new addition to the team's working practices. The notes documented the available servers considered useful in the context of test creation and maintenance, the configuration required to make them accessible from the contributors' development environments, and the situations in which their use was expected to provide the most benefit. These notes were intentionally kept lightweight and treated as a living reference, expected to be revised as the team's familiarity with these tools and their

[1] Model Context Protocol (MCP) servers are standardised interfaces that allow AI assistants to interact with external tools, services, and data sources. They expose capabilities — such as querying a file system, running commands, or calling an API — in a form that AI-powered development environments can invoke directly, enabling context-aware assistance during tasks such as test creation and code navigation.

associated workflows continued to develop.

4.2 Software & Hardware Architecture

This section presents the main architectural decisions made to optimise runner usage, restructure automated testing, and define the most reliable execution environment for the CI/CD workflow. Beyond reliability and resource efficiency, the architectural decisions described in this section were also guided by the goal of establishing an AI-ready execution environment — one in which test runs are containerised, the codebase is structured and parseable, and tooling can interface with modern AI-assisted workflows without requiring further architectural changes.

4.2.1 Hardware Resource Usage

The runners, being an essential hardware resource for the CI/CD pipeline to function, were one of the central topics in the first improvement meetings. Since the pipeline was triggered every time a developer pushed code to an MR, many unnecessary jobs were being executed, especially automated test battery runs, which were notably heavy and consumed a large amount of resources. As a result, automated test execution initially took between two and four times longer than expected, mainly due to queue times caused by full runner usage.

One solution that was tried was to have the QA assignee manually trigger the automated test job only when starting manual testing. In this way, unnecessary runs were avoided and queue times were reduced almost to zero. However, this solution was later abandoned because delaying the automated battery also delayed feedback to developers. In practice, this meant that a QA member could manually approve an MR and only afterwards discover, through a battery failure, that regressions existed in the platform and had not been detected during manual validation. In that situation, the approval would become invalid, the MR would have to return to development, and uncertainty would be created around the parts already tested by QA.

These downsides and the impact they had on the development process led to the conclusion that the more appropriate approach was to prioritise scaling hardware as the battery and resource consumption grew, addressing the problem at the root by proportionally allocating additional resources so that hardware would not bottleneck the workflow.

4.2.2 E2E Automated Test Battery

To understand the later restructuring of the automated test battery, it is first necessary to describe the process that originally led to the creation of end-to-end tests. When a client or company member detected a malfunction in the application, or when a feature was considered important enough to the overall platform behaviour to justify au-

tomated coverage, an issue was created together with a video showing the expected interaction. That issue was then added to the backlog of work requiring automated testing. A QA member would later pick one of these issues, usually according to priority, and translate the observed behaviour into an end-to-end test. A general representation of this process is shown in Figure 4.4.

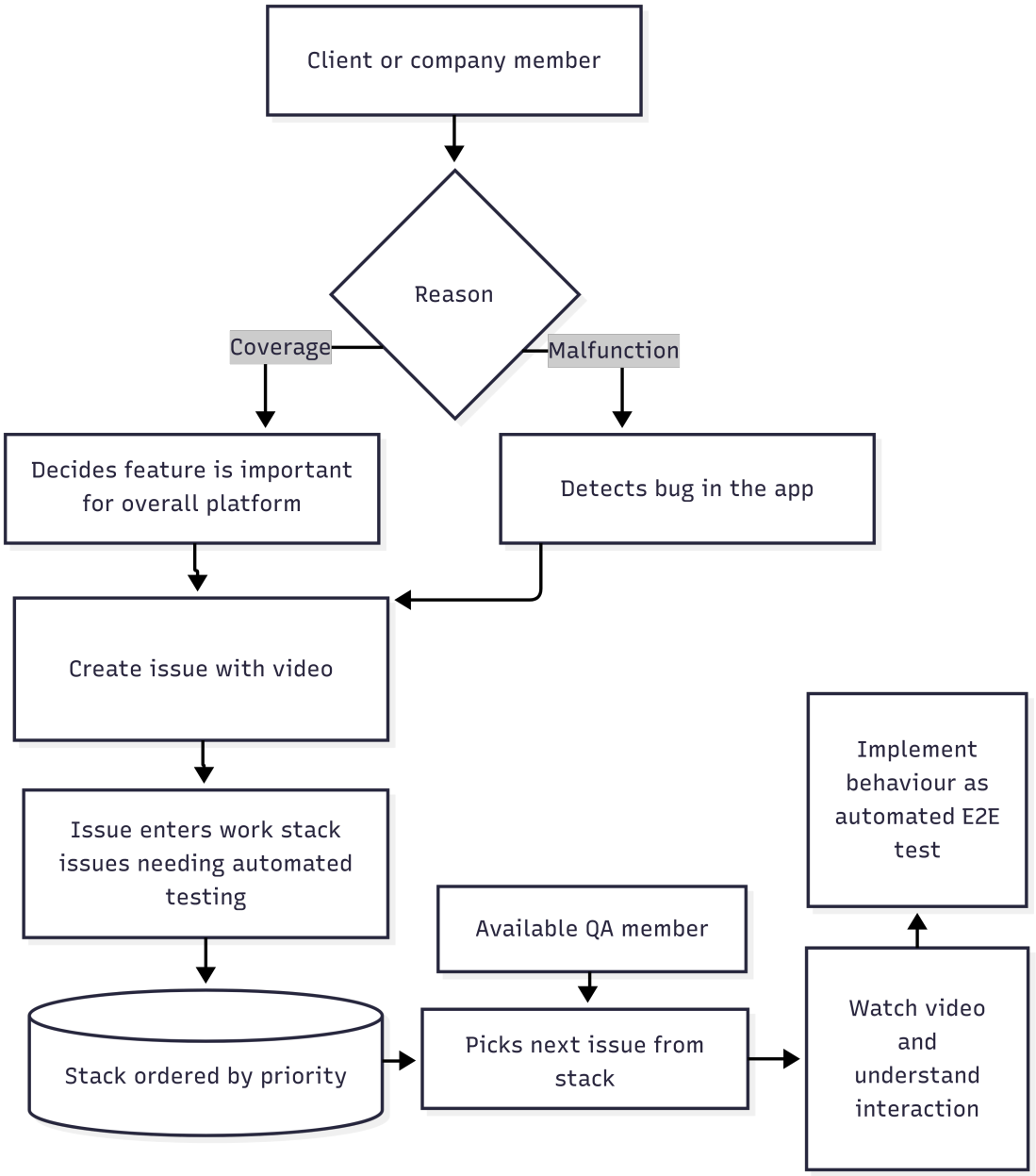

Figure 4.4: *Workflow for creating automated end-to-end tests from issues with demonstration videos and the QA backlog.*

This process guaranteed test coverage in the product areas considered more critical in case of failure, helping ensure that a real user could execute the intended actions in the platform and that the behaviour remained stable and as expected. It also supported a relatively direct implementation style, since the tests were created by reproducing ob-

servable user behaviour rather than by modelling deeper internal logic. As a result, the generated tests were generally easy to understand, because their purpose was closely tied to visible user interactions.

Upcoming Challenges and Disadvantages

However, as the company, the platform, and consequently the test battery grew, this model started to show significant limitations. End-to-end tests are inherently resource-intensive, and even small differences in platform behaviour could require two very similar tests that differed in only one or two commands while still containing dozens, and sometimes hundreds, of interaction steps. This redundancy reduced the scalability of the suite and consumed considerable time and hardware resources.

In addition, debugging failures in end-to-end tests became a substantial burden on developers. To understand the origin of a regression, the developer often needed a complete understanding of the full test flow and of all the user interactions represented in it. As the platform became more complex, the reference videos and the corresponding scenarios also became longer and harder to interpret. For these reasons, the original process increasingly showed the need for a revised approach better suited to the growing scale and complexity of the platform. These challenges and needs led to the creation of the structures and tests described in the Data and Test Restructuring section, and were specifically addressed by the introduction of the Unit and API Integration tests described below.

Automated Test Environment Choice and Dilemma

Another recurring discussion within the company, and one that remained active during the internship, concerned whether the automated test battery should run against the environment deployed by the pipeline or against an isolated environment provisioned specifically for testing. The standard practice was to split the tests and run them against environments built directly on the runners, and this was ultimately the option that was kept, consistent with the execution model illustrated in Figure 3.2.

The main alternative considered for improving pipeline efficiency was to use the deployed environment, hosted on an external cloud service, as the base for running the automated battery. In theory, this would reduce the use of internal hardware resources and remove one setup step from the pipeline, leading to faster execution and eliminating the possibility of failures related to environment creation. However, organisational experience and internal evaluation identified four main findings that led to the rejection of this proposal:

1. Problems related to local environment generation did not occur at a representative enough rate to justify replacing the existing approach.
2. Environment provisioning, build, and setup times represented only a small fraction of the total automated test job. At approximately two minutes, compared

with an average full-battery duration of around four hours, setup accounted for less than 1% of total execution time, remaining close to 0.8% on average.
3. Manual testing and general access to the deployed web environment could interfere with the seeded platform data on which the tests depended, causing accidental failures unrelated to the implementation under validation.
4. Most importantly, the transition would introduce additional vulnerable points, including network instability, high traffic, and dependence on external services that could become unavailable. These risks did not exist to the same extent when the same machine both provisioned the environment and executed the tests, making the isolated approach more secure and reliable.

Taken together, these disadvantages were considered more significant than the limited benefits expected from the change. For that reason, running the battery against isolated runner-provisioned environments was retained as the most reliable solution.

4.3 Data and Test Restructuring

This section describes the structural changes made to the test data and the automated test suite. The work spanned three interconnected areas: the introduction of new test types to rebalance the testing pyramid, the reorganisation of seed data, and the definition of test structure conventions to improve consistency and long-term maintainability.

4.3.1 New Types of Tests

Two new categories of automated tests were introduced as part of this restructuring effort, each targeting a different level of the testing pyramid and addressing a distinct limitation of the existing end-to-end battery.

API Integration Tests

The previously described challenges regarding the E2E battery prompted discussion in team meetings, leading to the decision to implement Unit and Integration Tests. In this section, we cover the arguments and decision process that led to the implementation of these tests. However, the technical details of how they were implemented and which tools were used mostly fall outside the intern's involvement and, therefore, the objectives of this report.

The first agreed priority was the implementation of integration tests, beginning with API tests. This ordering was deliberate: API tests target request handling and state updates, placing them at the layer closest to E2E tests in the testing hierarchy and making them the natural first step in progressively moving coverage down the testing pyramid. Furthermore, by validating the API layer directly, these tests offer more precise and explicit regression detection than E2E tests; failures surface immediately

at the contract boundary, with a clear signal and without requiring the full execution of a user flow to be observed. The decision stemmed from the observation that a significant number of existing E2E tests could be decomposed into lighter, more targeted tests, reducing redundancy and lowering overall execution time. Figure 4.5 illustrates the target state of the restructured suite.

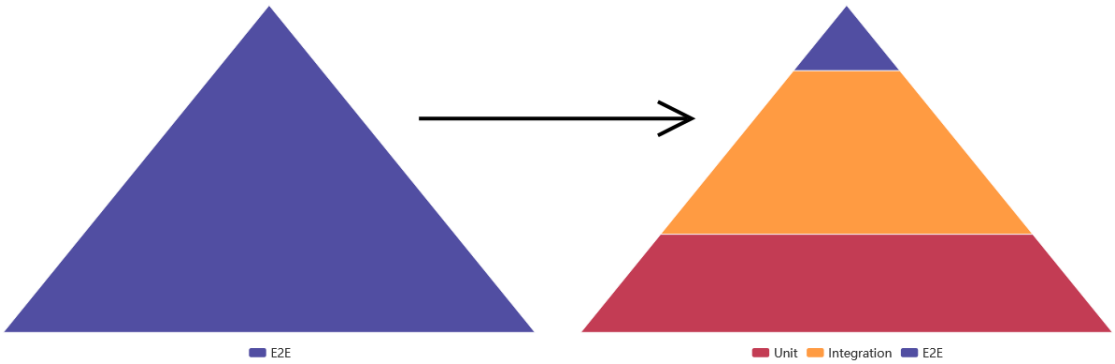

Figure 4.5: *Testing pyramid illustrating the target distribution of the restructured test suite.*

As E2E tests are gradually replaced by integration tests, which are lighter to execute and can run in parallel, this allows the battery to shrink in duration while growing in efficiency, and coverage metrics become more granular and actionable. Ideally, the suite should be composed of approximately 70% Unit tests, 20% Integration tests, and 10% E2E tests.

The API tests target the GraphQL backend, covering queries, mutations, and subscriptions, and are designed to validate the business logic underpinning the platform's core workflows. As part of the continuous integration pipeline, these tests generate coverage reports, offering visibility into which operations are exercised and where gaps remain. Additionally, the team identified performance API testing as a planned next step once functional coverage reaches a stable baseline, with the goal of ensuring response times comply with agreed service-level targets.

Unit Tests

The second priority was the development of Unit tests. These tests validate individual, isolated units of the platform, enabling behaviours that were previously exercised only through full E2E flows to be verified in a simpler and faster manner, without the overhead of a complete application setup. Together with integration tests, they form the foundation of a more balanced and scalable test suite. Of the various types requiring implementation (component, server, web, and Flutter unit tests), the team prioritised component tests first, with the remaining types falling outside the scope of this document. These tests are considerably less costly to write and maintain, as they exercise small pieces of code in isolation and offer greater efficiency relative to functional and E2E tests. These component tests would be implemented using Playwright and Storybook for screenshot testing, where the current implementation of a component across

different states is compared against a previously captured reference screenshot, detecting regressions in individual components through this visual comparison.

While the technical implementation was led by the more experienced developers, the intern conducted a structured analysis to identify which E2E tests should be migrated and in what order. Migration priority was determined by three criteria: the instability rate of each test, test behaviour and its execution time, with the highest priority assigned to tests that were simultaneously the most unstable and the most resource-intensive.

4.3.2 Seed Data Restructuring

It is important to note that the constant growth of the product transformed what had started as a simple test battery and set of seed data examples into complex systems of information and seeded data. The main driver was the expansion of the product with new tabs covering distinct functionalities, each requiring additional seeded information and new tests. Initially, the seeding process relied on three core files: one index file that called all the seeded data and executed all the seeding steps, one file containing the helper functions that aided the injection of information into the backend, and a queries file that held certain queries directly due to their low complexity. As the product grew, these three files became notably extensive, spanning multiple areas of the platform. This made them particularly difficult to read, as someone unfamiliar with these files would need to invest a considerable amount of time and pay careful attention to understand how all of the information was connected.

The solution that emerged was straightforward: create a dedicated folder and subdivide the index file into multiple files, one for each section (tab) of the platform, with the index file remaining as a central orchestrator that called all the other files in the established order, preserving the process structure without encapsulating the details. This allowed for easier navigation and development, with each section now clearly delimited, reducing the likelihood of unintentionally interfering with the seeded data of other platform areas.

The same approach was applied to the helpers: instead of a single file containing all platform helpers, multiple files were created, each corresponding to one section of the platform. Should a new feature be developed in a new or existing area of the platform, the developer could simply create a new file or extend the existing section file, respectively.

Lastly, all hardcoded queries were removed, retaining only the data structures for seed injection. Although this introduces an additional step of complexity, given the volume of seeded data and the likelihood that some structures would need to be seeded multiple times with minor variations, helpers were created and the relevant information was distributed across the files that resulted from splitting the main index file.

A further refinement involved the simplification of the existing seed data. Among the helpers and functions that supported the process of querying the backend to inject information, some functions existed solely to label and distinguish seeded data from non-seeded content. Functions that acted as intermediary steps, inserting identifiers into the queried material, were removed. An example would be a function that appended a string such as "- seeded for testing -" to the title of a seeded object. It was decided that this complexity should be removed and that it should be the responsibility of whoever authored the seed query to make clear, where necessary, that the information belonged to seeded objects. The benefit of this change was that seeded values appeared in the UI exactly as defined, without any intermediary modifications.

4.3.3 Test Structure and Conventions

Beyond the reorganisation of seed data, several conventions and structural changes were introduced to improve the consistency, independence, and navigability of the test suite. These changes covered variable naming, the relationship between tests and seed data, inter-test coupling, and the physical organisation of spec files within the repository.

Variable Naming Convention

One of the first implemented practices was to create a new centralised, standardised file with all the naming conventions for all the variables in E2E tests. Instead of declaring variables before each test, which would need to be redeclared multiple times across different test files, this centralised file contained the majority of common variables, providing the string values tests required. This allowed for more readable and consistent tests, as well as easier editing, since only one location needed to be updated. This was particularly useful given that the tests relied on the seed data, so both the seed data and the tests used the same file as a basis for nomenclature. When multiple variables were needed for the same object, such as the title and description of a project, the file contained a function that generated them with the same base name and an indexed number, so that they were easily identifiable as belonging to the same object. An extract of this file is shown in Figure 4.6.

For tests that required specific variables not shared with other tests, these variables were declared at the beginning of the test file itself. They also followed the same naming convention as those in the centralised file, so that there was a consistent naming standard across all tests, regardless of where they were declared. After these changes, the overall file structure and how the centralised file relates to the test specs can be seen in Figure 4.7.

```ts
// A helper to generate an indexed name: "Base 0"      You, 2 months ago • chore(tests
export const withIndex = (base: string, index: number) => `${base} ${index}`;

// A helper to generate an object of indexed names: { T0: "Base 0", T1: "Base 1"... }
const generateIndexedObject = (base: string, count: number, prefix: string) => {
  const obj: Record<string, string> = {};
  for (let i = 0; i < count; i++) {
    obj[`${prefix}${i}`] = withIndex(base, i);
  }
  return obj;
};

export const cypressTemplates = {
  ...generateIndexedObject('Cypress - IssueTemplate', 6, 'TEMPLATE'),
  ...generateIndexedObject('Cypress Group', 5, 'G'),
  ...generateIndexedObject('Cypress Task', 5, 'T'),

  INPUTS: {
    ...generateIndexedObject('Cypress Text Input', 5, 'TI'),
    ...generateIndexedObject('Cypress Selection Input', 5, 'SI'),
    ...generateIndexedObject('Cypress Number Input', 5, 'NI'),
  },
```

Figure 4.6: *Extract of the centralized variable naming convention file used across E2E tests.*

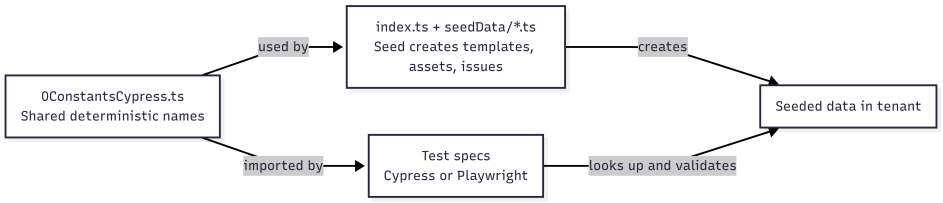

Figure 4.7: *Overview of the file structure showing how the centralized naming convention file relates to the test specs.*

Seed as a Base for Tests

The seed data was originally built as a tool to support QA manual testing, and while using the seed in test development was encouraged, it was not a standardised principle to expand the seed to accommodate the creation of different new tests. This meant that whenever a slightly different set of data was needed for a test, the developer would create it from scratch within the test spec, instead of using the seed as a base and adding the necessary information to it. This created redundancy, as over time some tests started using the same long setups of data that were built in the specs, but could not be reused due to the difficulty in understanding what was seeded without running the full spec. This was because parsing through the test setup required the reader to distinguish between UI interaction commands and the API calls that actually created the test data.

This process unintentionally made the tests more difficult to maintain, as they became longer, with more failure points and sometimes made it difficult to understand what the spec specifically aimed to test. Having the information setup present in the seed allows both QA and other developers to grasp the objective of the test with more ease, as fewer commands are executed, reducing each spec to its test logic only, free from data setup commands.

A last but highly important step of this process was reaching the conclusion that there was no need to keep redundant test data teardown in the test specs. As previously established, the specs were built with a prescribed order that consisted of: 1. building the necessary information for the tests; 2. executing the test behaviour; 3. tearing down the created and edited data back to an original clean state. However, earlier decisions in the development process established that when running in the pipelines with the automated test environment, environments were always built from a clean state for each test run, which means that any issues that could arise from rerunning tests that depend on seeded data are inherently avoided, as the environment is always reset and reseeded before each run. This is because it is easier and faster to delete and reseed the whole environment than to wait for the full individual teardown of test information. The other concern that could exist is that the tests could interfere with each other; however, it has been established as a core principle of test development that tests should not rely on shared data between them, and the seed contains the necessary information independently for each one of them.

Strengthening Test Independence

In the course of this work, a new challenge emerged: as each spec covered a specific part of the platform and the battery grew, tests within the same spec had slowly become so interconnected that it became notably hard to pick a single test and run it individually. This happened because as specs grew, whenever a new test was added to the battery, it had to account for the state established by the preceding tests in the same spec. If any developer or QA member tried to run the test individually, it might fail because it was expecting the environment state created by the previous test runs. Two causes mainly led to this: 1. using locators by page position instead of data-test-id attributes; 2. relying on information that was created only after previous tests in the spec were executed. The first mostly occurred due to the absence of data-test-id attributes on page components, an overlooked step of development that later led QA to use the page location instead of the component identifier. The second happened due to the internal objective of not making longer specs: before the seed-based approach was in place, reusing information already created by other tests in the spec meant shorter specs but introduced structural coupling. However, this caused the test behaviour to resemble a single long complex test made up of smaller interconnected steps, rather than independent linear tests each focused on a specific feature of the platform area.

To combat this, a new principle was established that no spec should contain more than 2 or 3 tests, and ideally each spec should be limited to 1 test, unless reasonable justification was provided. This principle, combined with the process of moving the information setup to the seed data and the removal of redundant teardown, made it possible to tackle this challenge in an easier and more structured way, making tests more independent. As the test setup was moved to the seed, it became clearer what data was needed to execute each test, and it was possible to eliminate the existing inter-

dependencies. Since the seed is always executed in full before the run, all tests would find all the necessary information already in place, as illustrated in Figure 4.8.

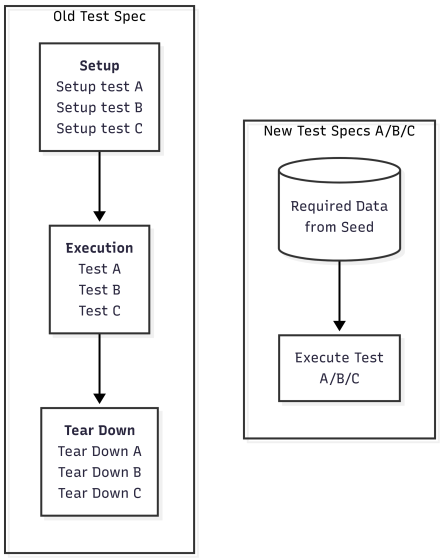

Figure 4.8: *Comparison between the old spec structure, with multiple tests sharing a single setup and teardown, and the new structure, where each spec contains a single test that retrieves its data directly from the seed.*

Splitting specs that contained multiple tests, sometimes more than 50 per spec, meant having 50 times more specs, which could later create challenges when navigating the test battery folder. To address this, it was decided that naming specs after the platform area would be replaced by organising them into a folder tree following the same naming scheme, making navigation easier and providing a more natural location for adding new tests. This approach also raised the possibility of subdividing each folder into subfolders, as the platform grew and many new features would start appearing, causing the test battery to expand. Instead of having a single spec named for a dedicated area of the platform, a folder with the same name now contained the multiple specs that covered all the different tests. Should the platform later add subdivisions to a given area, it was now possible to group the existing tests into a subfolder and create a new folder for the new subdivision, extending the folder tree accordingly, as shown in Figure 4.9.

E2E vs API Analysis

As these changes were being implemented, the appearance of the API tests presented, in parallel, the opportunity to refine the test battery further. The existing E2E test suite contained tests that could be further simplified and moved down the pyramid as described earlier in the test types section. Since API tests are lighter and faster, the need arose to conduct an analysis on which E2E tests to keep and which to migrate down the pyramid. It was also noted that this should be the first task in the whole process, as adapting E2E tests to the new structure, adjusting naming conventions and

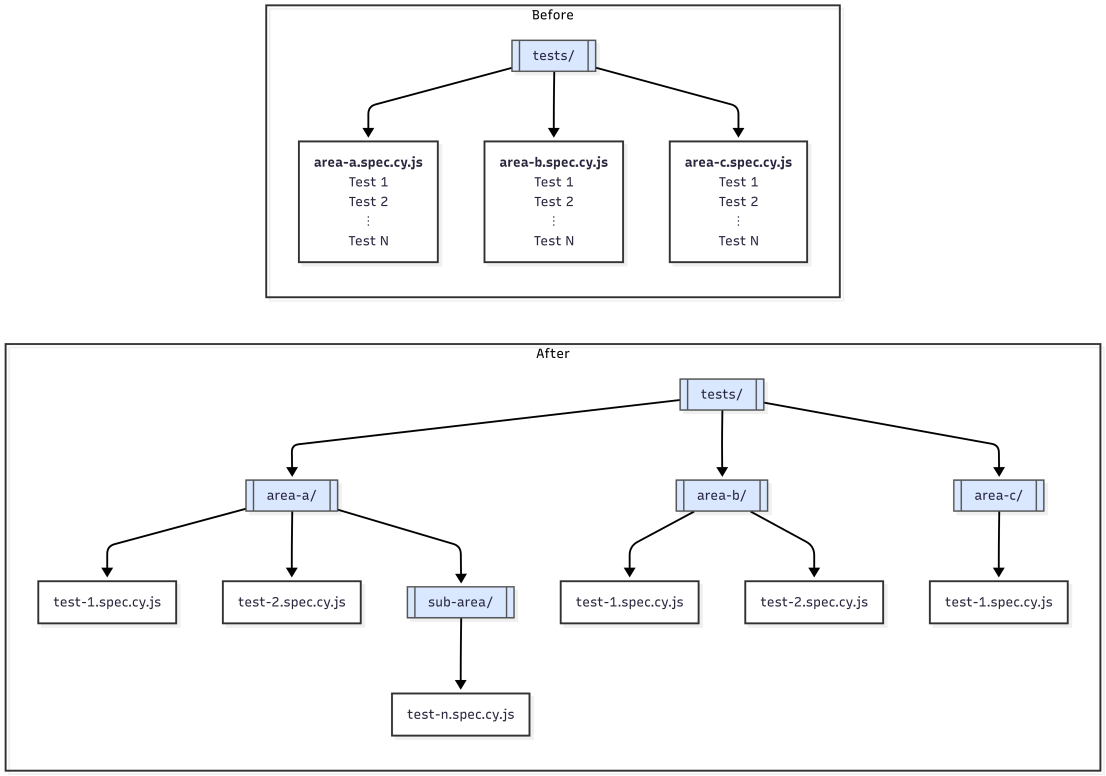

Figure 4.9: *Transition from a flat collection of spec files, each covering an entire platform area, to a folder tree where each folder groups the individual specs for that area and can be further subdivided into subfolders as the platform grows.*

seed data for them, would be an inefficient effort if these tests were subsequently to be migrated to API tests. With this in mind, an analysis was conducted to firstly determine which tests should be adapted to the new structure and which should not be touched, as they would just be converted to API tests and subsequently removed from the E2E test battery.

The criteria were straightforward: any E2E tests whose main execution path primarily consisted of validating API and data behaviour, confirming whether the information had been successfully sent, edited, received, or deleted, would be migrated to API tests. The UI coverage lost in this migration would later be covered by the creation of the new proposed component (unit) tests, as described in the test types subsection.

E2E separation from UI

The original placement of the E2E tests within the UI folder was a natural initial decision: since end-to-end tests exercise the application through its user interface, co-locating them with the frontend codebase seemed intuitive. However, as the battery grew in both size and complexity, this arrangement started showing its limitations.

From a structural standpoint, placing the tests inside the UI folder implicitly tied their perceived ownership to the frontend. Any contributor unfamiliar with this convention would need to navigate into the UI folder to locate the test suite, reducing its

discoverability and, consequently, the likelihood of it being consulted or maintained across the broader team.

More critically, the E2E test battery had by this point evolved into a concern that was functionally independent from the UI source code. The tests did not need to be compiled, bundled, or deployed alongside the frontend application; they operated as a separate process that consumed the running application as a black box. Their dependencies, configuration files, and execution scripts were therefore distinct from those of the UI, and grouping them under the same directory created a misleading sense of coupling between the two.

Moving the E2E tests to a dedicated top-level directory resolved both concerns. It gave the battery a clearly demarcated space in the repository, making it immediately visible and accessible to any contributor regardless of their primary area of work. It also reinforced the understanding that the test suite was an independent workstream with its own maintenance cycle, decoupled from changes to the frontend codebase. This separation aligned with the broader effort to elevate the QA battery to a first-class concern within the development process, ensuring that its structure reflected the role it had come to play in the team's workflow.

4.3.4 Parallelization in Pipeline Execution

Another topic of discussion was the parallelization of the test execution in the CI/CD pipeline. At the time the internship began, the existing configuration split the test battery automatically into two equal halves, each executed in a separate environment provisioned by the same runner. This allowed the two groups of specs to run concurrently, reducing the total pipeline duration compared to a fully sequential run.

Previous attempts had been made to push this further by splitting the battery into a greater number of parts, each running in its own isolated environment. However, this approach did not yield the expected improvements. Because the environments were all raised on the same physical machine, dividing execution into more parallel branches meant dividing the same pool of hardware resources among a greater number of simultaneously running processes. With the specs being long and not particularly efficient at the time, the overhead introduced by provisioning and running additional environments outweighed any reduction in per-branch duration, resulting in slower overall execution rather than faster. This highlighted an important constraint: parallelism on shared hardware does not scale indefinitely, and the gains from adding more concurrent branches diminish as the resource contention increases.

An additional structural limitation was identified in the tooling itself. Cypress, which was the framework in use at the time, does not natively support test parallelisation within a single installation. Distributing test execution across multiple workers requires Cypress Cloud, a paid external service. This dependency made any significant improvement to the parallelisation strategy contingent on either adopting external

infrastructure or migrating to a different tooling approach.

Given these constraints, the conclusion reached was to leave the existing split between two environments unchanged. Altering the parallelisation configuration without first addressing the underlying inefficiencies in spec structure would not produce meaningful gains. It could also introduce additional instability into the pipeline. The situation was, however, explicitly noted as a scalability concern: as the test battery continued to grow, the inability to distribute execution more efficiently would place an increasing burden on pipeline duration. This observation was later taken into account as part of the broader discussion around tooling migration covered in the following section.

4.4 Tools and Migrations

This section covers the tooling changes and migrations carried out during the internship. These ranged from targeted automation scripts that reduced repetitive manual steps to a full framework migration that addressed the structural limitations of the existing test suite. Each change responded to a specific friction point identified in the course of the broader improvement process.

4.4.1 Local Test Environment Automation

Before any automated test could be executed locally, several sequential setup steps had to be completed: starting the Docker containers, bringing up the backend server, running the seed data injection, and finally starting the UI. Each of these steps had to be performed in the correct order, and failure at any stage required manual intervention before the process could continue. While this sequence was well-defined, its manual execution represented a significant friction point, particularly for contributors who did not interact with the test environment on a daily basis.

This friction had a direct impact on local test validation. When an automated test failed in the CI/CD pipeline, the natural course of action was to reproduce the failure locally in order to isolate its cause. However, the effort required to bring up a local environment from scratch was substantial enough to discourage this practice, especially under time pressure. If the test needed to be re-run after a code correction, as is common during debugging, the entire setup sequence had to be repeated, since no mechanism existed to reset only the affected data while leaving the rest of the environment intact.

To address this, a shell script was developed to automate the full local environment provisioning process. The script encapsulates the complete setup sequence, executing each step in the prescribed order without requiring manual intervention. When invoked against an environment that is already running, it first performs a full teardown, clearing all previously seeded data, before re-executing the setup from the beginning.

This behaviour ensures that the environment is always in a known, clean state at the start of any test run, regardless of what may have occurred in previous sessions.

The practical effect of this change was a significant reduction in the effort required to run tests locally. Contributors no longer needed to remember the exact sequence of setup commands or manage partial environment states manually. Developers encountering pipeline failures in their MRs could reproduce and investigate the issue locally by simply executing the script, and repeat the process as many times as necessary with the same single command. This lowered the barrier to local test validation and encouraged a broader adoption of the practice, particularly among developers who would previously have relied solely on the CI/CD pipeline output to diagnose test failures.

4.4.2 Battery migration to Playwright

The migration to Playwright was one of the most significant technical challenges, being the result of a structured evaluation process driven by a persistent instability in the existing battery and by the framework's growing scalability constraints. It was carried out gradually to ensure that automated coverage remained uninterrupted throughout the transition.

Existing Challenges & Migration Considerations

It is necessary to understand that the initial trigger for this process came naturally from ongoing work and analysis on the existing battery. To identify and address instabilities that arose in the test battery with ongoing development, an analysis of automated battery run reports was necessary. This analysis aimed to pinpoint which areas of the battery suffered from the most instability and why. This process, although adapted in the course of the internship to provide more findings and structure, such as the construction of Python scripts for analysing and compiling results from multiple test runs, was not new to the company.

The analysis presented here, conducted over three months of test runs while the improvements described in this report were being implemented in parallel, demonstrated that different tests started becoming unstable over time and that, despite having more concentrated instability in some tests, a vast majority of tests failed at spaced intervals, with approximately 80% of tests failing at least once per month. This led to the conclusion that the source would have to be located in the generic behaviour and helper functions and the way Cypress executed them, rather than in any specific interactions of the platform. One solution that had been introduced prior to the internship to address this instability was an `interceptGraphQL()` function that awaited backend responses before continuing to the next step of the testing blocks, counteracting Cypress's asynchronous behaviour. To clarify, Cypress commands are asynchronous and non-blocking: they are queued and executed in a strict sequence rather than immediately when called, with no native mechanism for awaiting responses in-

line. While this considerably improved instability in some tests, others continued to fail, and the problem persisted and grew with the battery despite the mitigation. The failures were characterised as intermittent: the local environment presented no connection or network problems, and the failures were never reproducible manually by users, even though they were consistently reproducible across various environments and machines running the Cypress automated test suite. It was therefore concluded that this asynchronism and the absence of explicit wait mechanisms were the most likely root cause of instability.

While investigating solutions and considering the broader challenges associated with the testing framework, such as the parallelisation constraints discussed in the preceding section, Playwright emerged as a viable alternative. It offered native parallelism without dependency on external services, in contrast to Cypress's reliance on a paid cloud service for the same purpose. As the project and the battery continued to grow, this limitation had become an increasingly significant constraint on scalability.

The decision to migrate was not, however, taken lightly. The existing battery had reached considerable size and was underpinned by a substantial body of helper functions developed over the course of the project. As established in Chapter 3, the evaluation of Playwright was approached not as an impulsive replacement but as a structured technical validation exercise intended to determine whether the advantages observed in theory would hold against the team's specific infrastructure and requirements. Migrating a suite of this scale away from an active QA pipeline represented a non-trivial risk: any disruption to automated coverage during the transition would have a direct impact on the development cycle. Commitment to a full migration was therefore withheld until the pilot had demonstrated that the transition was operationally viable and brought successful results.

The evaluation identified three specific advantages that aligned directly with the team's requirements. First, the mentioned Playwright native parallelism across multiple workers. Secondly, Playwright's execution model is based on native async/await patterns that allow test steps to be expressed and awaited inline in a sequential, readable manner. This stands in direct contrast to Cypress's command-queue mechanism, the behaviour that had been identified as the primary source of instability in the existing battery. Thirdly, Playwright offers native compatibility with MCP server tooling, enabling AI-assisted workflows in which agents can interact directly with the browser automation layer during test creation and maintenance, an aspect that heavily aligned with the broader infrastructure direction described later in this chapter and reduced the barrier to contribution for team members less familiar with the test suite. This third point is specially important as diminishing the cost of development for future tests meant the cost of migration would be balanced out in the future, making it specially desired, as the earlier the migration would start, the earlier the returning benefits would be achieved.

Pilot and Progressive Migration

Before committing to a full migration, a structured pilot was designed around the instability findings already gathered. Using previous pipeline failure reports as a data source, individual tests were evaluated against two criteria: their instability rate across runs and their execution time. From this evaluation, a selection of the highest-priority tests was made, those that were simultaneously the most unstable and the most resource-intensive in Cypress were chosen as the subjects of the initial migration to Playwright. The idea was straight-forward: if the tests that had shown the most consistent instability in Cypress exhibited no instability once implemented in Playwright, this would validate the framework's suitability for the battery and of the hypothesis that the instability was attributable to Cypress's execution model rather than to the test scenarios themselves.

This pilot consisted of a spec with 32 tests, confirming this outcome. The migrated tests, which had been among the most problematic in the Cypress suite, ran without the intermittent failures that had characterised their behaviour under Cypress, demonstrating that Playwright's native async/await execution model resolved the instability in precisely the cases where Cypress had consistently failed. This result validated both the technical argument for migration and the operational feasibility of carrying it out without disrupting the active QA pipeline.

Following this successful outcome, the migration was taken forward gradually. The Playwright battery is being built up incrementally as specs are migrated, while the Cypress battery remains operational in parallel during the transition period, ensuring that automated coverage is maintained without interruption at any stage of the development cycle.

4.4.3 Feedback Recording Software

Before the introduction of any dedicated tooling, the process for reporting bugs or suggesting improvements to the platform relied on a combination of external software and manual steps. When a team member identified an anomalous behaviour or wanted to propose a change, the expected approach was to capture the occurrence using a general-purpose screen recording application, and subsequently upload the resulting video file. The recording would then be attached as a comment on the MR associated with the feature under development, or used as supporting material when opening a new GitLab issue. While this workflow allowed team members to convey visual context alongside written descriptions, it was time consuming and introduced friction into the reporting process.

The dependency on an external application meant that contributors had to switch context, configure and operate a separate tool, managing and uploading the resulting video files manually before submitting feedback. For team members who were not

already familiar with OBS, the setup overhead was particularly discouraging. More broadly, the effort required at each step: capturing, uploading, attaching, and composing a written description, made casual or low-priority observations less likely to be submitted at all, reducing the overall volume of internal feedback reaching the development team.

To address this, a feedback recording tool was created in the form of a browser extension. The extension was designed to integrate directly into the team's existing GitLab-based workflow, removing the dependency on any external application or manual file management.

The extension operates through a straightforward interface. On first use, a one-time configuration step links the extension to the team member's GitLab account, enabling it to interact with the platform directly from the browser (Figure 4.10). Once configured, a single button initiates a screen recording of the active browser window. When the recording is stopped, the user is presented with a choice of three submission types (Figure 4.11): adding a comment to an existing MR, submitting an improvement suggestion, or filing a bug report. For the first option, the user provides only the MR's numeric identifier, and the extension handles the submission automatically. For improvement suggestions and bug reports, the form is pre-populated with a structured text template, which the contributor replaces with the specific details of their observation. Template texts are available in all three cases to include a descriptive comment alongside the recorded video.

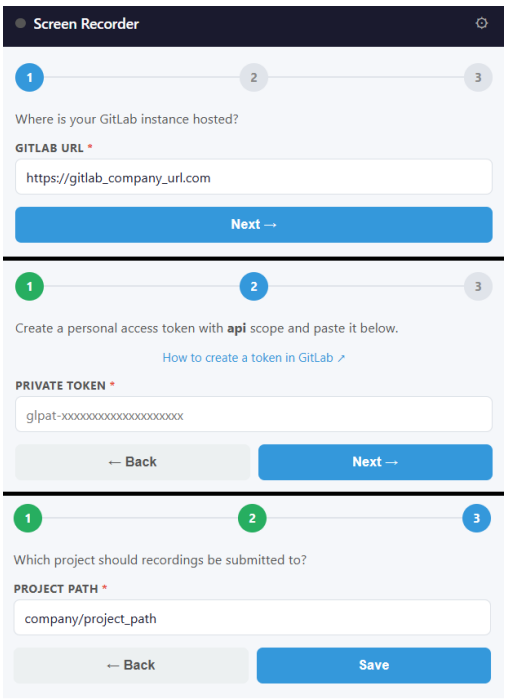

Figure 4.10: *One-time configuration screen for linking the extension to a GitLab account.*

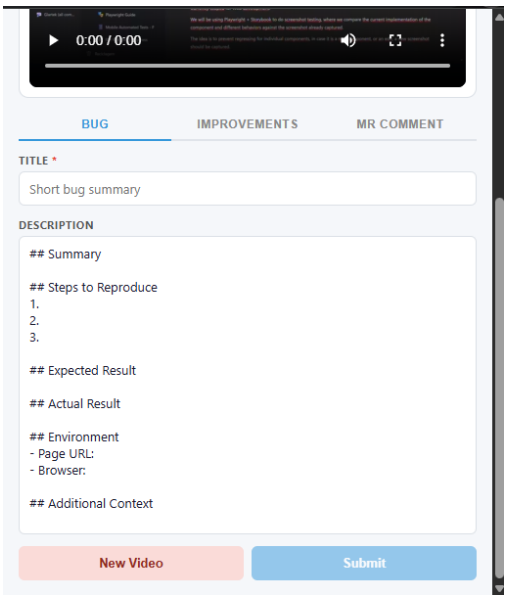

Figure 4.11: *Submission form presented after a recording is stopped, offering options for MR comment, improvement suggestion, or bug report.*

This design addressed the two main limitations of the previous approach. The recording and submission steps were consolidated into a single, browser-native workflow, eliminating the dependency on OBS and the manual overhead associated with file upload and issue creation. The structured templates also standardised the format of submitted reports, ensuring that all feedback reached the development team in a consistent, actionable form regardless of who submitted it.

The practical effect reduced the effort required to submit feedback, the extension lowered the threshold for participation and encouraged contributions from team members who had previously found the manual process too burdensome at times. The resulting increase in reporting activity provided greater convenience and a broader shift toward more systematic quality observations within the organisation, reinforcing the idea that identifying and communicating platform issues are a shared responsibility rather than the exclusive domain of the QA team.

4.4.4 Final Workflow-Infrastructure and AI-ready Environment

One architectural consequence of the decisions described in this chapter is that the resulting environment is well-positioned to accommodate AI-driven workflows without requiring further re-engineering. The containerised execution model introduced for both the CI/CD pipeline and the local development script ensures that test runs are reproducible and operable without human intervention, a property that automated agents depend on to interact with the environment reliably. At the codebase level, the adoption of standardised naming conventions, centralised variable declarations, and modular seed data structures means that the test suite is parseable and navigable by automated tooling in a way that ad hoc, undocumented configurations are not. Playwright's support for a Model Context Protocol server further extends this compatibility, allowing AI agents to interface directly with the browser automation layer and issue commands against a running test environment through a structured protocol. Taken together, these properties: reproducible execution, structured codebase and an open automation interface, place the infrastructure in a state where AI-assisted testing workflows can be introduced incrementally, as the foundations are already compatible with the patterns described in the literature on AI-driven continuous testing [2, 26].

4.5 Contributions

The process described in this chapter involved multiple stakeholders across the organisation, including developers, team leads, product and design representatives, and the QA team itself. However, the intern's specific contributions, while often embedded in broader collaborative efforts, can be clearly identified. Tables 4.2 through 4.7 summarise each contribution grouped by chapter area, along with the type of participation and relevant contextual detail.

Participation types are defined as follows: *Proposed* indicates that the intern originated the idea; *Implemented* that the intern directly built or created the artefact; *Authored* that the intern wrote documentation or written content; *Analysed* that the intern conducted a structured analysis informing a decision; and *Collaborated* that the intern contributed alongside other team members toward a shared outcome.

Table 4.2: *Contributions — Workflow*

Contribution	Participation	Details
QA self-assignment to MRs	Proposed & Implemented	Prevented duplicate work and improved task visibility and filtering within the team.
`qa manually tested` label	Proposed & Implemented	Signals that manual approval is complete and the MR is awaiting pipeline confirmation.
`qa manual testing skipped` label	Proposed & Implemented	Covers minor bug-fix MRs exempt from full manual testing.
`automated tests - needs improvement` label	Proposed & Implemented	Links known unstable automated tests to their originating GitLab issue for faster failure triage.
Reviewer role expansion	Collaborated	Established an earlier code-quality review layer before MRs reach QA, reducing avoidable developer–QA iterations.
Test responsibility matrix	Collaborated	Defined and documented the distribution of responsibilities across roles for each new test type introduced in the restructuring effort, covering acceptance criteria ownership, test development, and review.
Confirmation modal mock-up	Proposed	Designed to reinforce developer responsibility when setting the `Testing required` label; not yet implemented due to higher-priority infrastructure migration work.
Label rename proposal	Collaborated	Renaming `Testing required` to `Tested by developer` to make ownership expectations explicit at the point of label assignment; not yet implemented.

Table 4.3: *Contributions — Workflow: Documentation*

Contribution	Participation	Details
QA Rulebook rewrite	Authored	Replaced an outdated, extensive document with a compact, cross-team-accessible rulebook repositioned as the central QA reference.
Development process guides	Authored	Guides covering QA procedures and test-creation workflows, aimed at onboarding support and cross-team comprehension.
Tool usage guides	Authored	Setup and usage documentation supporting the adoption of new tooling, including Playwright.
MCP server usage notes	Proposed & Authored	Produced lightweight documentation on available MCP servers, their configuration, and the contexts in which their use was expected to provide the most benefit during test creation and maintenance.
Documentation pruning	Proposed & Implemented	Archived or reduced the visibility of outdated content covering replaced processes, keeping the documentation space aligned with current working procedures.

Table 4.4: *Contributions — Software & Hardware Architecture*

Contribution	Participation	Details
Manual trigger trial and evaluation	Proposed & Analysed	Explored as a mechanism to reduce unnecessary pipeline runs; subsequently abandoned because it delayed automated feedback to developers.
Decision to scale hardware resources	Collaborated	Contributed to the case for addressing runner bottlenecks at the root rather than through workflow workarounds.
Runner-isolated environment retention	Analysed & Collaborated	Contributed to the analysis that led to rejecting the cloud-hosted environment option for automated battery execution.

Table 4.5: *Contributions — Data and Test Restructuring*

Contribution	Participation	Details
API integration test priority analysis	Analysed	Identified existing E2E tests that could be decomposed into lighter, more targeted API-layer tests, informing the restructuring roadmap.
E2E-to-unit migration priority ranking	Analysed	Evaluated migration candidates by instability rate, execution time, and resource cost to establish migration order.
Seed data index file split	Proposed & Implemented	Divided a monolithic seed file into per-platform-section files, retaining the index as a central orchestrator.
Helper files reorganisation	Proposed & Implemented	Replaced a single all-platform helper file with section-based files, reducing the risk of cross-area interference during development.
Hardcoded query removal	Proposed & Implemented	Replaced inline queries with structured data declarations and reusable helper functions, improving maintainability.
Seed data simplification	Proposed & Implemented	Removed intermediary naming functions; seeded values now appear in the UI exactly as declared without any intermediate modification.
Seed-as-base test approach	Proposed & Implemented	Standardised the practice of using seed data as the sole source of test data, removing data-creation commands from specs and reducing each spec to its test logic only.
Post-test teardown removal	Proposed & Implemented	Established that per-spec data teardown was redundant given that environments are always rebuilt from a clean state; removed teardown blocks to reduce spec length and eliminate additional failure points.

Table 4.6: *Contributions — Data and Test Restructuring: Test Structure and Conventions*

Contribution	Participation	Details
Variable naming convention	Proposed & Implemented	Created a centralised file defining naming standards for all E2E test variables, shared across specs and seed data to ensure consistency and single-point editing.
One-test-per-spec principle	Proposed & Implemented	Introduced the convention of limiting each spec to one or two tests to eliminate inter-test state dependencies and improve individual runnability.
Folder tree restructuring	Proposed & Implemented	Replaced area-named spec files with a nested folder structure mirroring the platform hierarchy, improving navigability and providing a scalable location for future test additions.
E2E vs. API test scope analysis	Analysed	Determined which existing E2E tests should be adapted to the new structure and which should instead be migrated to API tests, establishing this as the required first step to avoid investing effort in adapting tests that were subsequently to be removed from the E2E battery.
E2E separation from UI folder	Proposed & Collaborated	Moved the E2E test battery to a dedicated top-level repository directory, decoupling it from the frontend codebase and making it accessible as an independent workstream.
Parallelisation constraint analysis	Analysed	Evaluated the limitations of expanding parallel spec execution on shared hardware and documented the scalability concern, informing the subsequent tooling evaluation.

Table 4.7: *Contributions — Tools and Migrations*

Contribution	Participation	Details
Local environment automation script	Proposed & Implemented	Developed a shell script encapsulating the full local test environment setup and teardown sequence, reducing manual effort and lowering the barrier to local test validation for all contributors.
Playwright Evaluation & Migration	Analysed & Implemented	Investigated Playwright as a migration target for the E2E battery, examining advantages including native parallelism without dependency on external services, and MCP and AI tooling integration.
Pipeline analysis Python scripts	Proposed & Implemented	Developed Python scripts to aggregate and structure results from multiple automated test runs, enabling systematic identification of instability patterns across the battery.
Instability analysis and test cleanup	Analysed & Implemented	Examined pipeline failure reports to identify tests with high instability rates, determined their root causes, and removed or corrected them as part of the migration preparation, reducing noise in the battery and improving the reliability of pipeline results.
Feedback recording browser extension	Proposed & Collaborated	Built a browser extension integrating screen recording directly into the GitLab workflow, replacing the external OBS dependency and standardising the feedback submission format with structured templates.

5 Results

The changes described in the Implementation chapter were not introduced in a single coordinated release but were deployed progressively, with each intervention building on the stability of the preceding one. This chapter presents the outcomes of those interventions, following the same structural organisation. Where quantitative data were available, concrete measurements are provided. Where the effects were primarily organisational or qualitative in nature, the results are presented as observed outcomes and, where relevant, as team feedback.

5.1 Workflow

The workflow changes took effect progressively throughout the internship period. Unlike the infrastructure and tooling changes that followed, their impact was largely organisational and was therefore assessed through observed team behaviour and feedback rather than through measurable metrics. Despite the absence of collected quantitative metrics, these were among the most impactful changes overall: a frictionless environment leading to better communication and fewer handover failures contributed to a more productive and healthier working environment within the company, a shared sentiment reflected in daily meetings and sprint planning sessions.

5.1.1 Task Separation and Handover

The introduction of QA self-assignment and the expanded label set resolved ambiguities that had previously created friction in the testing queue. With QA members assigning themselves to MRs on pickup, duplicate testing efforts ceased to occur and the visible state of each MR in the queue became more informative at a glance. These changes were implemented right at the beginning of the internship, so there was not enough data collected to represent the previous state of the process, but no instances of duplicated work were recorded throughout the course of the internship, achieving

the intended result.

The expanded label set covered scenarios that had previously been handled more informally. The `qa manually tested` label made it possible to distinguish MRs that had been manually approved but were still awaiting automated pipeline confirmation, a state that had previously been invisible in the workflow. Feedback from the development team indicated a unanimous improvement in the clarity of handover states, with developers and QA members expressing greater confidence in reading the current status of an MR without needing to consult comments or teammates.

5.1.2 Responsibility Ownership

The introduction of the expanded reviewer role added a structured quality gate earlier in the development cycle. Defects that would previously have been identified only during manual QA, such as logic inconsistencies, implementation gaps and structural concerns began to be surfaced before MRs reached the testing queue. This reduced the developer–QA iteration loop, making it less likely for most of the avoidable rework cycles.

Developer feedback on this change was consistently positive. The clearer distribution of responsibilities, as summarised in Table 4.1, was reported to reduce ambiguity about who was expected to validate what, particularly for the new test types introduced alongside the restructuring effort. These clarifications also helped reduce frustration: with clearly defined and written responsibility ownership, each team member knew where their work ended and where that of others began. This reduced friction arising from messy handovers that had previously resulted from unclear task ownership.

Workflow Shift and New Development Cycle

The most structurally significant result of the workflow changes was the repositioning of the automated pipeline as a prerequisite for manual QA rather than a parallel or later activity. This shift, illustrated in Figure 4.3, was made possible by the improvements to the automated battery described in the subsequent sections. Its practical effect was to ensure that QA manual validation only began after the pipeline had already confirmed the absence of regressions, a condition that had not been consistently enforced in the prior workflow. This was only possible after having created a sense of trust and reliability on the automated test battery itself, by mitigating and progressively eliminating the sources of instability in the automated test runs.

This change reduced one class of iteration: cases in which QA approved an MR manually only for a subsequent pipeline failure to reopen it, rendering the previous manual validation effort invalid, as by rule manual validation is required after every change to the implementation. Although no formal cycle-time measurements were captured, the team's qualitative assessment was that the revised workflow reduced

unnecessary back-and-forth and made the overall progression from development to approval more linear.

5.1.3 Documentation

The documentation produced during this period was evaluated primarily against its adoption and utility as a working reference. The rewritten QA Rulebook was reinforced as the primary onboarding document for new QA members and served as a reference point for developers from other areas seeking to understand testing procedures. The testing framework guides not only reduced the time required to bring new QA members up to speed with the test infrastructure but also reduced the reliance of the onboarding QA members on senior employees for clarifications, giving them more time for their specialized tasks, as the structural decisions behind the battery's organisation were now documented in a single accessible location rather than residing implicitly in the knowledge of the senior members who had originally built it.

The tool usage and infrastructure guides extended working knowledge of the new framework across the team and outside the group directly involved in the migration. Documentation pruning removed material covering replaced processes, reducing the risk that outdated guidance would be followed inadvertently and making the remaining content more likely to be consulted as an authoritative reference.

The newly built guides, being easier to read, navigate, and understand, made them more attractive as a reference, allowing knowledge to be spread more efficiently and with less effort, contributing to the development of the team as a whole. Figure 5.1 shows the resulting documentation hub, where each question a contributor is likely to ask is addressed by a targeted guide, providing a single structured entry point into the QA knowledge base.

5.2 Software & Hardware Architecture

This section presents the outcomes of the architectural decisions described in the Implementation chapter, covering the resolution of runner resource contention, the instability baseline established from the Cypress battery analysis, and the validation of the isolated test execution environment.

5.2.1 Hardware Resource Usage

The scaling of hardware resources addressed the queue-time problem at its root. Following the provisioning of additional runners, pipeline execution time returned to expected durations, with automated test jobs running within their normal range rather than being extended by resource contention. The manual trigger trial, which had reduced queue times to near zero but at the cost of delayed automated feedback to developers, was superseded by this more permanent solution, restoring the earlier-feedback

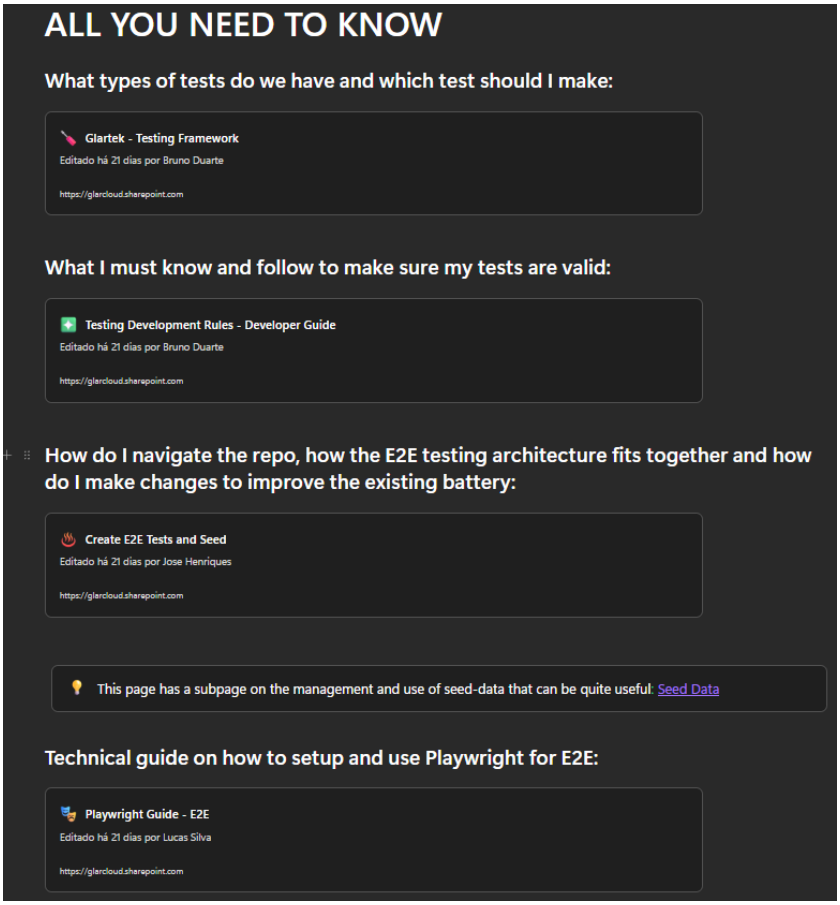

Figure 5.1: *Documentation hub consolidating all QA guides under a structured set of questions, serving as the primary entry point for both QA members and developers.*

benefit while eliminating the runner bottleneck.

5.2.2 E2E Automated Test Battery Instability

To contextualise the results of the tooling migration described in the following section, it is necessary to document the instability baseline of the Cypress battery. An analysis of 237 CI pipeline runs conducted between December 2025 and February 2026 identified 1,233 failure instances across 159 unique tests, yielding an average of 520.3 failure instances per 100 parsed runs. Only 38 of the 237 runs, around 16% of the total, completed without any test failure. Of these failures, 88.5% were classified as timeouts, consistent with the asynchronous execution model issues described in the Implementation chapter and with the `interceptGraphQL()` workaround that had been introduced precisely to mitigate this behaviour.

This baseline established that instability was a pervasive property of the battery rather than a problem localised to a small number of tests, and provided the quantitative foundation for the migration evaluation described in Section 5.4.2. Figure 5.2 charts the scale and evolution of this problem across three measurement periods: failure instances outnumbered parsed runs by a factor of roughly five in the initial baseline, with the ratio progressively improving through the later periods as the restructuring and migration work took effect.

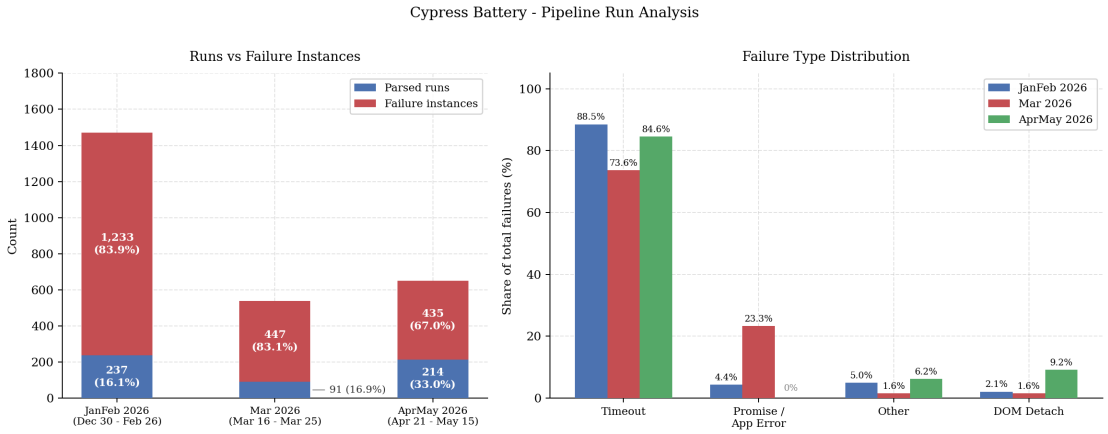

Figure 5.2: *Cypress battery analysis across three measurement periods. Left: stacked comparison of parsed runs (blue, bottom) and failure instances (red, top); the ratio fell from roughly five-to-one in January–February (237 runs, 1,233 failures) and March (91 runs, 447 failures) to approximately two-to-one in April–May (214 runs, 435 failures). Right: failure type distribution across all three periods, showing the March spike in Promise/App errors (23.3%) fully resolved in April–May (0%), while DOM Detach errors increased (1.6% → 9.2%) and the timeout share recovered to 84.6%.*

5.2.3 Automated Test Environment Choice

The decision to retain runner-provisioned isolated environments was validated throughout the observation period. No failures attributable to environment provisioning or setup occurred at a rate that would have justified the architectural risks associated with the proposed cloud-hosted alternative, confirming that the primary

sources of test instability lay in the framework execution model rather than in environment creation. This confirmed that the reliability of locally hosted, runner-provisioned environments was a critical requirement for the development process, as minimising externally introduced failures was one of the core goals of the infrastructure design.

5.3 Data and Test Restructuring

This section presents the outcomes of the structural changes made to the test data and the automated test suite. The results span the introduction of new test types, the reorganisation of seed data, and the conventions applied to improve consistency and independence across the battery.

5.3.1 New Types of Tests

The introduction of API integration tests established a new coverage layer between E2E and unit tests, enabling validations that would previously have required full application setups to instead target the GraphQL layer directly. Component tests were introduced using Playwright and Storybook, providing visual regression coverage at the component level. The addition of these test types began to shift the suite towards the target distribution described in Figure 4.5, though the transition remained in progress at the time this report was written. However, these changes allowed for the start of trimming of the E2E battery, migrating E2E tests that were better suited and redistributed through the API and component layer. The substitution of each E2E test for component and API tests provided benefits in terms of pipeline efficiency, as E2E represent the most hardware resource intensive layer of tests, and failure isolation, that helped better identify regressions. This enabled regression debugging work to proceed much faster, as by identifying the source of failure from API and component tests is easier than navigating the long artifact with walk-through videos and reports created after E2E test failures. These artefacts were storage-heavy, reaching up to 1 GB of downloaded material per failure, and difficult to navigate and draw conclusions from for both QA and developers. Compared to this, the new artifacts consisted mostly of simple lines of code that directly pointed at the cause of failure.

5.3.2 Seed Data Restructuring

The reorganisation of the seed data files produced improvements that were primarily qualitative in nature but significant for long-term maintainability. The subdivision of the monolithic index file into per-section files, and the corresponding reorganisation of helper functions, reduced the risk of cross-area interference during test development and made the seed navigable to contributors without prior detailed knowledge of its structure. The simplification of naming functions, by removing the intermediary steps that had appended identifiers to seeded values, reduced processing overhead in the seed injection pipeline and ensured that seeded data appeared in the UI exactly as de-

clared, eliminating a class of discrepancy that had previously complicated test debugging. These changes produce no directly measurable metric improvements in isolation, but generally improve the developer experience, enabling faster implementations and reducing the likelihood of errors in future development.

5.3.3 Test Structure and Conventions

The structural changes to the test suite produced the most directly measurable results of the restructuring effort. The Actions platform area served as the representative case. Prior to restructuring, the spec covering this area contained 82 tests, of which 71 were active and 11 were pending, meaning they were skipped due to instability, and a a complete run averaged 11 minutes and 25 seconds, measured across three runs on different runners and merge requests at the time of the start of implementation. After the seed-as-base approach was applied, inter-test dependencies were removed, and the one-test-per-spec principle was enforced, the restructured battery for the same area contained 31 tests, all active and passing with none pending, averaging 6 minutes and 28 seconds per run. This represents a 62% reduction in test count, a 43% reduction in execution time, and the elimination of all pending tests, which had represented dead weight in the suite without contributing to coverage. Figures 5.3 and 5.4 show examples of run summaries for the original and restructured specs respectively, and Figure 5.5 provides a visual comparison of test count and health across all three stages.

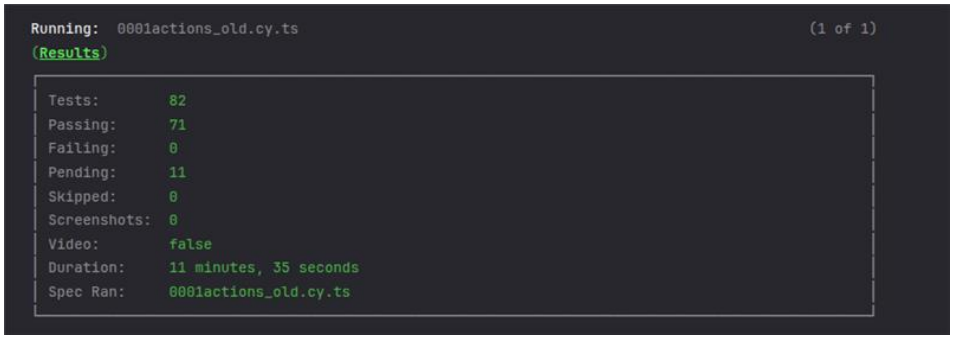

Figure 5.3: *Example run summary for the original Actions spec: 82 tests (71 passing, 11 pending). The average run duration across three measured runs was 11 minutes and 25 seconds.*

Figure 5.4: *Example run summary for the restructured Actions spec: 31 tests (all passing, 0 pending). The average run duration was 6 minutes and 28 seconds.*

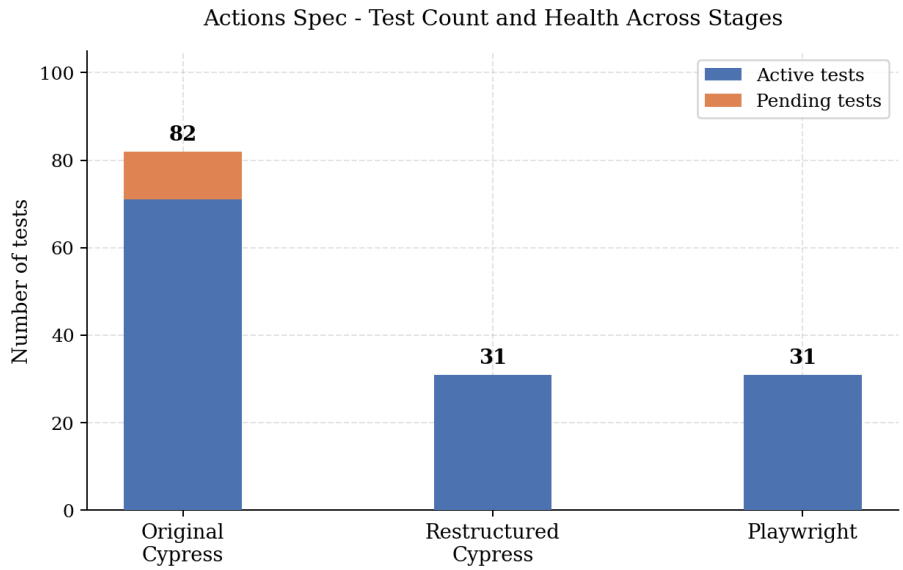

Figure 5.5: *Test count and health across the three stages of the Actions spec: original Cypress (82 tests, 11 pending), restructured Cypress (31 tests, 0 pending), and Playwright (31 tests, 0 pending).*

Across the broader battery, the analysis conducted as part of the E2E versus API scope review identified approximately 200 tests, which represent around 22% of the total E2E battery, as candidates for removal or migration to the seed data layer. These tests were primarily characterised by containing data-creation or teardown logic that either duplicated seed data, introduced unnecessary inter-test coupling, or validated behaviour more appropriately covered by API-level tests. The restructuring of these tests was ongoing at the time this report was written, with the Actions area serving as a completed reference case for the approach.

Parallelisation

The parallelisation constraint analysis confirmed that expanding the number of parallel execution branches on shared hardware would not yield meaningful gains under the existing spec structure. This conclusion, reached through direct experimentation, was later validated indirectly by the test restructuring work: as specs were shortened and inter-test coupling was removed, individual tests became lighter and more suitable for distribution across workers, a prerequisite for effective parallelisation that had not previously been met.

5.4 Tools and Migrations

This section presents the outcomes of the tooling changes and migrations carried out during the internship, from the local environment automation script through the Playwright migration pilot to the feedback recording extension and the resulting AI-ready infrastructure.

5.4.1 Local Test Environment Automation

The local environment automation script reduced the full setup sequence to a single command. Contributors encountering pipeline failures could reproduce and investigate them locally without managing partial environment states or recalling the correct sequence of setup steps, and could repeat the process as many times as necessary with the same invocation. The practical effect was a lower barrier to local test validation and broader engagement with the test environment across the team, particularly among developers who had previously relied solely on CI pipeline output to diagnose failures.

5.4.2 Battery Migration to Playwright

The Playwright migration pilot produced the clearest quantitative result of the tooling changes. The Actions spec described in the previous section, now restructured to 31 tests and selected precisely because its constituent tests had been among the most consistently unstable in Cypress, was migrated to Playwright and executed without any of the intermittent failures that had characterised its behaviour in the prior framework. This outcome directly validated the hypothesis advanced in the Implementation chapter: that the instability was attributable to Cypress's execution model rather than to the test scenarios and consequently platform interactions themselves. Beyond stability, the migration also yielded a further reduction in execution time: the same spec that had averaged 6 minutes and 28 seconds under the restructured Cypress configuration completed in 4 minutes and 48 seconds under Playwright, representing an additional 26% reduction on top of the gains already achieved through restructuring. Figure 5.6 charts the cumulative progression in run duration across all three stages.

Two subsequent measurement periods, summarised in Figure 5.2, document a progressive reduction in battery instability. The March 2026 period recorded 491.2 failure instances per 100 runs, a marginal reduction from the January–February baseline of 520.3; however, this comparison must be interpreted with caution, as the March dataset contained 753 files that could not be parsed due to malformed XML output, meaning the actual improvement may be greater than the raw figures suggest. The April–May 2026 period showed substantially greater improvement: failure instances fell to 203.3 per 100 runs, a 58.6% reduction from March, and the proportion of failure-free runs increased from 12.1% to 42.1%. The Promise/App error category, which had spiked to 23.3% of failures in March, was fully resolved in AprMay (0%), confirming that the underlying application error handling issue had been addressed. Taken together, the three periods show a greater than 60% reduction in failure rate from the January–February baseline to April–May, with the proportion of failure-free runs increasing from 16% to 42%.

The progressive migration continued beyond the pilot, with the Playwright battery being built up incrementally while the Cypress battery remained operational and reduced gradually in parallel, ensuring uninterrupted automated coverage throughout

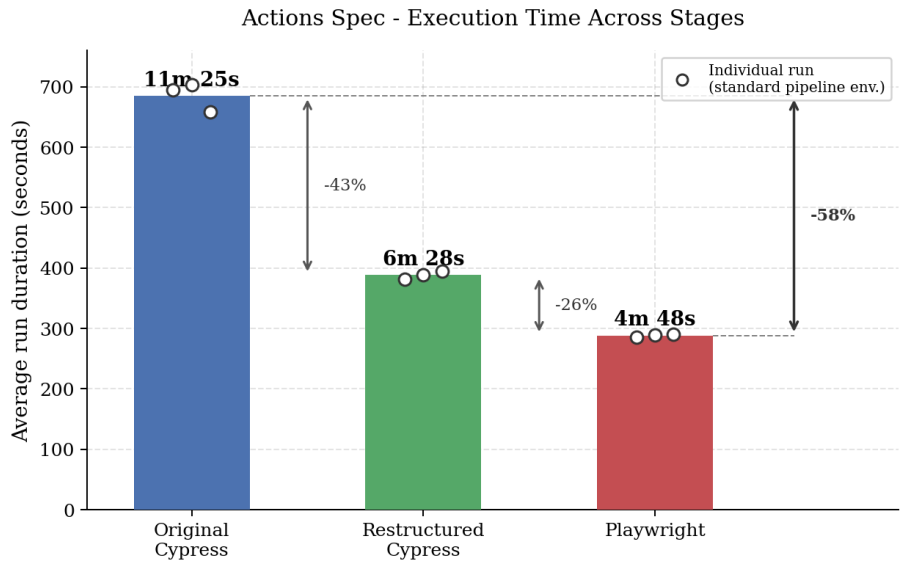

Figure 5.6: *Average run duration of the Actions spec across the three stages: original Cypress (11m 25s), restructured Cypress (6m 28s), and Playwright (4m 48s). Hollow circles represent individual measured runs in a standard pipeline environment. Step reductions are annotated between adjacent bars (−43% and −26%); the right-hand arrow shows the overall reduction from original Cypress to Playwright (−58%).*

the transition.

5.4.3 Feedback Recording Software

The browser extension eliminated the dependency on external screen recording software, consolidating the recording, annotation, and submission steps into a single native browser workflow. The structured submission templates standardised the format of feedback reports arriving at the development team, reducing the interpretive effort required to act on them. This led to an increase of around 15% in manually opened issues and suggestions compared to the month prior to the implementation of the extension. However, this information needs to be understood carefully as it might have other factors contributing to it such as new feature implementations and suggestions for new stories. Team members who had previously found the external tool workflow too burdensome under time pressure reported in meeting points a lower threshold for submitting observations, resulting in a broader and more consistent flow of internal quality feedback.

5.4.4 Final Workflow-Infrastructure and AI-ready Environment

The architectural properties established across the interventions described in this chapter, containerised and reproducible execution, structured and parseable codebase, and native support for MCP-based tool integration through Playwright and other tools, positioned the infrastructure to support AI-assisted workflows without requiring further re-engineering. This represents a structural result rather than a directly measurable

outcome: the conditions for incremental adoption of AI-driven testing workflows are now in place, consistent with the direction described in the literature on AI-assisted continuous testing [2, 26].

5.5 Future Work

Several areas identified during the course of this work represent natural next steps beyond the scope of this report.

The most immediate continuation of the restructuring effort is the completion of the seed migration for the approximately 200 tests identified as candidates. As the Actions area demonstrates, this process is well-defined and produces measurable gains in both execution time and test independence; extending it across the remaining identified cases is therefore a direct continuation of work already validated in practice.

A complementary direction is the continued expansion of the API integration and unit test suite, with the explicit goal of relieving pressure on the E2E battery. The review conducted during this internship identified a significant proportion of existing E2E tests whose constituent actions were primarily verifying API responses, data mutations, or isolated unit behaviour rather than genuine end-to-end user journeys. These tests were present in the E2E battery for lack of a lower-level alternative at the time of writing rather than because E2E was the appropriate coverage layer. As the API and unit layers mature, these cases should be progressively rewritten at the correct level and removed from the E2E battery, reducing E2E execution time, hardware pressure, and failure surface while preserving and improving overall coverage.

The Playwright migration represents a longer-term commitment. With the pilot confirming the technical viability of the approach and the progressive migration under way, the goal is to complete the transition across the full battery and eventually decommission the Cypress suite. Alongside this, coverage metrics for the E2E and API integration tests have not yet been formalised; establishing coverage thresholds and generating regular reports would represent a further maturation step for the battery.

At the workflow level, the confirmation modal described in the Implementation chapter has not yet been implemented due to competing infrastructure priorities. Its adoption would strengthen responsibility ownership by making the developer commitment to MR readiness explicit at the point of label assignment.

Building on the structured codebase and MCP-capable infrastructure, a further direction is the development of reusable agent skills grounded in the team's internal documentation. Such skills could encode established migration patterns such as the Cypress-to-Playwright conversion rules or the seed-based restructuring procedures, surfacing them as on-demand, context-aware tools. Applied to the E2E battery, a documentation-grounded agent could identify tests whose assertions are confined to

API responses or isolated state changes and generate candidate rewrites at the appropriate lower layer, accelerating the progressive redistribution of coverage described above. Because the rationale and patterns are captured in documentation rather than embedded in opaque prompts, the skills remain auditable and maintainable as conventions evolve, and can be refined incrementally as the team accumulates experience with AI-assisted workflows.

Finally, the AI-ready infrastructure established during this internship provides the foundation for AI-assisted test creation and maintenance workflows, but these have not yet been formalised into the team's standard practices. The groundwork is in place; incremental adoption as the team's familiarity with the tooling develops represents the logical progression.

6

Conclusion

This chapter brings documentation of this journey to a close. From the far-reaching objectives defined at the beginning, one conclusion stands clear: by the end of this internship, the processes were in a significantly better state than at the outset. Even though developer satisfaction metrics were not registered throughout the changes described here, there was a clear feeling of improvement, coming from a more accessible and welcoming environment. The quantified metrics showed promising progress across all aspects they covered, from test speed and stability to ease of use and scalability.

From the general goal of achieving a better infrastructure with more containerised processes, less waste, more maintainable code, tools, and documentation, one thing became very clear: this was the start of what is and must remain a continuous process. This demonstrates a need for continuous oversight and analysis as products evolve, incorporating new tools and procedures that may require rebuilding and redesigning processes. It also becomes clear that the earlier these analyses and improvements are carried out, the greater the benefit and the less the waste, avoiding the need for drastic redesigns later, when the system can no longer absorb accumulated changes without a full overhaul.

Building this document while living through this great experience also brought many lessons, both as an engineer and as a human being. It was possible to encounter the challenges associated with bringing theoretical knowledge to real-world companies and people, each with their own unique situations and constraints. It was nevertheless an outstanding opportunity to navigate all these dimensions while building a more resilient company and product, maintaining as a core value the respect for the needs and values of each individual involved. Perhaps one of the greatest lessons drawn from this experience relates to the importance of individuals' alignment with group goals, where the company represents the group of people oriented toward the same objective. This reflects the reality that making processes and systems better hinges on

centering them around the individuals involved, bringing everyone together, defining transparent objectives, and ensuring everyone benefits from moving toward the shared path to success.

As for Glartek, the host organisation at the centre of this work, I conclude with confidence and gratitude, knowing they are sailing towards a brighter, successful future, guided by strong principles that create internal value for their teams and external value for their clients and anyone who benefits from their solutions. This reminds us that in a world where change is the only constant and innovation emerges at every turn, those who are willing to embrace change and adapt accordingly will surely be rewarded with lasting success. It is my core belief that those who leverage scientific and technological progress for the sake of world sustainability and societal quality-of-life improvements will thrive as key drivers of humanity's future.

This page intentionally left blank.

Bibliography

[1] Vamsi Krishna Thatikonda. "Beyond the Buzz: A Journey Through CI/CD Principles and Best Practices". In: *European Journal of Theoretical and Applied Sciences* 1.5 (2023), pp. 334–340. DOI: 10.59324/ejtas.2023.1(5).24. URL: https://doi.org/10.59324/ejtas.2023.1(5).24.

[2] Ada John, Isreal John, and Trump Dion. "Integrating AI-Driven Test Case Optimization into Continuous Integration/Continuous Delivery (CI/CD) Pipeline". In: (May 2025).

[3] Eliezio Soares et al. "The effects of continuous integration on software development: a systematic literature review". In: *Empirical Software Engineering* 27 (2022), p. 78. DOI: 10.1007/s10664-021-10114-1. URL: https://doi.org/10.1007/s10664-021-10114-1.

[4] Shujun Huang and Sebastian Proksch. "A Taxonomy of Contextual Factors in Continuous Integration Processes". In: *IEEE Transactions on Software Engineering* 51.7 (2025). DOI: 10.1109/TSE.2025.3572382. URL: https://doi.org/10.1109/TSE.2025.3572382.

[5] Pooya Rostami Mazrae et al. "On the usage, co-usage and migration of CI/CD tools: A qualitative analysis". In: *Empirical Software Engineering* 28 (2023), p. 52. DOI: 10.1007/s10664-022-10285-5. URL: https://doi.org/10.1007/s10664-022-10285-5.

[6] Runxiang Cheng et al. "Revisiting Test-Case Prioritization on Long-Running Test Suites". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. ACM, 2024, pp. 1–13. DOI: 10.1145/3650212.3680307. URL: https://doi.org/10.1145/3650212.3680307.

[7] Bohan Liu et al. "The Why, When, What, and How About Predictive Continuous Integration: A Simulation-Based Investigation". In: *IEEE Transactions on Software Engineering* 49.12 (2023). DOI: 10.1109/TSE.2023.3330510. URL: https://doi.org/10.1109/TSE.2023.3330510.

[8] Islem Bouzenia and Michael Pradel. "Resource Usage and Optimization Opportunities in Workflows of GitHub Actions". In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024)*. ACM, 2024. DOI: 10.1145/3597503.3623303. URL: https://doi.org/10.1145/3597503.3623303.

[9] Maria Manuela Boto Abrantes. "Quality Assurance Automation". MA thesis. Faculdade de Ciencias e Tecnologia, Universidade de Coimbra, Sept. 2020.

[10] Henri Aïdasso, Mohammed Sayagh, and Francis Bordeleau. "Build Optimization: A Systematic Literature Review". In: *ACM Computing Surveys* 58.1 (2025), Article 12. DOI: 10.1145/3757912. URL: https://doi.org/10.1145/3757912.

[11] David Jorge Garcia Mendes. "Automated Testing for Provisioning Systems of Complex Cloud Products". MA thesis. NOVA University Lisbon, Sept. 2019.

[12] Selenium Project. *Selenium – Web Browser Automation*. 2026. URL: https://www.selenium.dev/ (visited on 2026-03-02).

[13] Cypress.io. *Continuous Integration with Cypress*. 2026. URL: https://docs.cypress.io/app/continuous-integration/overview (visited on 2026-02-21).

[14] Playwright.dev. *Continuous Integration*. 2026. URL: https://playwright.dev/docs/ci (visited on 2026-02-21).

[15] Jashn Jain. *Cypress Market Share in 2026: npm, GitHub, and Enterprise Data*. Feb. 2026. URL: https://testdino.com/blog/cypress-market-share/ (visited on 2026-03-02).

[16] Microsoft. *Playwright Documentation*. 2026. URL: https://playwright.dev/docs/intro (visited on 2026-02-21).

[17] Playwright.dev. *Auto-waiting*. 2026. URL: https://playwright.dev/docs/actionability (visited on 2026-02-21).

[18] Cypress.io. *Cypress Documentation*. 2026. URL: https://docs.cypress.io/ (visited on 2026-02-21).

[19] Cypress.io. *Retry-ability*. 2026. URL: https://docs.cypress.io/guides/core-concepts/retry-ability (visited on 2026-02-21).

[20] Hugo Feitosa de Figueiredo et al. "Addressing the Synchronization Challenge in Cypress End-to-End Tests". In: *Anais do Simposio Brasileiro de Engenharia de Software (SBES)*. 2024, pp. 92–102. DOI: 10.5753/sbes.2024.3298. URL: https://doi.org/10.5753/sbes.2024.3298.

[21] Henri Aïdasso, Francis Bordeleau, and Ali Tizghadam. "On the Diagnosis of Flaky Job Failures: Understanding and Prioritizing Failure Categories". In: *arXiv preprint arXiv:2501.04976* (2025). URL: https://arxiv.org/abs/2501.04976.

[22] Henri Aïdasso, Francis Bordeleau, and Ali Tizghadam. "Efficient Detection of Intermittent Job Failures Using Few-Shot Learning". In: *arXiv preprint arXiv:2507.04173* (2025). URL: https://arxiv.org/abs/2507.04173.

[23] Florent Moriconi. "Improving Software Development Life Cycle Using Data-Driven Approaches". English. NNT: 2024SORUS164; HAL Id: tel-04700138. PhD thesis. Sorbonne Université, 2024. URL: https://theses.hal.science/tel-04700138.

[24] Henri Aïdasso, Francis Bordeleau, and Ali Tizghadam. "On the Illusion of Success: An Empirical Study of Build Reruns and Silent Failures in Industrial CI". In: *arXiv preprint arXiv:2509.14347* (2025). URL: https://arxiv.org/abs/2509.14347.

[25] Shiva Krishna Kodithyala. "Smart Test Selection in CI/CD: Optimizing Pipeline Efficiency". In: *Journal of Computer Science and Technology Studies* 7.4 (2025), pp. 289–297. DOI: 10.32996/jcsts.2025.7.4.33. URL: https://doi.org/10.32996/jcsts.2025.7.4.33.

[26] Bharath Chandra Vadde and Vamshi Bharath Munagandla. "Integrating AI-Driven Continuous Testing in DevOps for Enhanced Software Quality". In: *Revista de Inteligencia Artificial en Medicina* 14.1 (2023), pp. 505–515.

[27] Nuno André Faustino Bernardino. "Desenvolvimento de Ferramentas e Práticas para uma Implementação Efetiva de DevSecOps". MA thesis. Instituto Politécnico de Leiria, June 2024.

[28] Gabriel Dario Chanchay Tituaña. "Continuous Integration Methodology Implementation and Comparison". MA thesis. Instituto Politécnico de Leiria, Sept. 2021.

[29] James P. Womack and Daniel T. Jones. *Lean Thinking: Banish Waste and Create Wealth in Your Corporation*. 2nd. Free Press, 2003. ISBN: 9780743249270.

[30] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003. ISBN: 9780321150783.

This page intentionally left blank.